可下载教学资料

http://www.tup.tsinghua.edu.cn

高等学校教材

计算机科学与技术

# 数据结构算法解析

高一凡 著

清华大学出版社

高等学校教材·计算机科学与技术

# 数据结构算法解析

高一凡 著

清 华 大 学 出 版 社
北 京

**内 容 简 介**

本书为严蔚敏、吴伟民编著的《数据结构(C语言版)》(清华大学出版社出版，本书将其简称为教科书)的学习辅导书。主要内容包括教科书中各主要数据存储结构的基本操作函数、调用这些基本操作的主程序和程序运行结果以及教科书中各主要数据存储结构的图示。

本书结合存储结构和算法，配合大量的图示，对于一些较难理解的算法，还配有文字说明。

本书所有程序均在计算机上运行通过，这些程序可通过清华大学出版社的网站下载。

本书适用于使用严蔚敏、吴伟民编著的《数据结构(C语言版)》作教材的高等学校学生和自学者，也可供使用其他《数据结构》教材者和软件编程人员参考，同时也是考研很好的参考书。



**图书在版编目(CIP)数据**

数据结构算法解析/高一凡著. —北京：清华大学出版社，2008.2
(高等学校教材·计算机科学与技术)
ISBN 978-7-302-15879-0

Ⅰ. 数…　Ⅱ. 高…　Ⅲ. ①数据结构—高等学校—教材 ②算法分析—高等学校—教材
Ⅳ. TP311.12

中国版本图书馆CIP数据核字(2007)第119312号

**责任编辑**：郑寅堃　张为民
**责任校对**：李建庄
**责任印制**：

**出版发行**：清华大学出版社　　**地　址**：北京清华大学学研大厦A座
http://www.tup.com.cn　　**邮　编**：100084
c-service@tup.tsinghua.edu.cn
**社 总 机**：010-62770175　　**邮购热线**：010-62786544
**投稿咨询**：010-62772015　　**客户服务**：010-62776969
**印 刷 者**：
**装 订 者**：
**经　销**：全国新华书店
**开　本**：185×260　**印张**：22　**字数**：531千字
**版　次**：2008年2月第1版　**印次**：2008年2月第1次印刷
**印　数**：1～　000
**定　价**：　.00元

---

高等学校教材·计算机科学与技术

# 编审委员会成员

（按地区排序）

| | | |
|---|---|---|
| 南京邮电学院 | 朱秀昌 | 教授 |
| 苏州大学 | 龚声蓉 | 教授 |
| 江苏大学 | 宋余庆 | 教授 |
| 武汉大学 | 何炎祥 | 教授 |
| 华中科技大学 | 刘乐善 | 教授 |
| 中南财经政法大学 | 刘腾红 | 教授 |
| 华中师范大学 | 王林平 | 副教授 |
| | 魏开平 | 副教授 |
| | 叶俊民 | 教授 |
| 国防科技大学 | 赵克佳 | 教授 |
| | 肖　侬 | 副教授 |
| 中南大学 | 陈松乔 | 教授 |
| | 刘卫国 | 教授 |
| 湖南大学 | 林亚平 | 教授 |
| | 邹北骥 | 教授 |
| 西安交通大学 | 沈钧毅 | 教授 |
| | 齐　勇 | 教授 |
| 长安大学 | 巨永峰 | 教授 |
| 西安石油学院 | 方　明 | 教授 |
| 西安邮电学院 | 陈莉君 | 副教授 |
| 哈尔滨工业大学 | 郭茂祖 | 教授 |
| 吉林大学 | 徐一平 | 教授 |
| | 毕　强 | 教授 |
| 长春工程学院 | 沙胜贤 | 教授 |
| 山东大学 | 孟祥旭 | 教授 |
| | 郝兴伟 | 教授 |
| 山东科技大学 | 郑永果 | 教授 |
| 中山大学 | 潘小轰 | 教授 |
| 厦门大学 | 冯少荣 | 教授 |
| 福州大学 | 林世平 | 副教授 |
| 云南大学 | 刘惟一 | 教授 |
| 重庆邮电学院 | 王国胤 | 教授 |
| 西南交通大学 | 杨　燕 | 副教授 |

# 出版说明

改革开放以来，特别是党的十五大以来，我国教育事业取得了举世瞩目的辉煌成就，高等教育实现了历史性的跨越，已由精英教育阶段进入国际公认的大众化教育阶段。在质量不断提高的基础上，高等教育规模取得如此快速的发展，创造了世界教育发展史上的奇迹。当前，教育工作既面临着千载难逢的良好机遇，同时也面临着前所未有的严峻挑战。社会不断增长的高等教育需求同教育供给特别是优质教育供给不足的矛盾，是现阶段教育发展面临的基本矛盾。

教育部一直十分重视高等教育质量工作。2001 年 8 月，教育部下发了《关于加强高等学校本科教学工作，提高教学质量的若干意见》，提出了十二条加强本科教学工作提高教学质量的措施和意见。2003 年 6 月和 2004 年 2 月，教育部分别下发了《关于启动高等学校教学质量与教学改革工程精品课程建设工作的通知》和《教育部实施精品课程建设提高高校教学质量和人才培养质量》文件，指出“高等学校教学质量和教学改革工程”是教育部正在制定的《2003—2007 年教育振兴行动计划》的重要组成部分，精品课程建设是“质量工程”的重要内容之一。教育部计划用五年时间（2003—2007 年）建设 1500 门国家级精品课程，利用现代化的教育信息技术手段将精品课程的相关内容上网并免费开放，以实现优质教学资源共享，提高高等学校教学质量和人才培养质量。

为了深入贯彻落实教育部《关于加强高等学校本科教学工作，提高教学质量的若干意见》精神，紧密配合教育部已经启动的“高等学校教学质量与教学改革工程精品课程建设工作”，在有关专家、教授的倡议和有关部门的大力支持下，我们组织并成立了“清华大学出版社教材编审委员会”（以下简称“编委会”），旨在配合教育部制定精品课程教材的出版规划，讨论并实施精品课程教材的编写与出版工作。“编委会”成员皆来自全国各类高等学校教学与科研第一线的骨干教师，其中许多教师为各校相关院、系主管教学的院长或系主任。

按照教育部的要求，“编委会”一致认为，精品课程的建设工作从开始就要坚持高标准、严要求，处于一个比较高的起点上；精品课程教材应该能够反映各高校教学改革与课程建设的需要，要有特色风格、有创新性（新体系、新内容、新手段、新思路，教材的内容体系有较高的科学创新、技术创新和理念创新的含量）、先进性（对原有的学科体系有实质性的改革和发展、顺应并符合新世纪教学发展的规律、代表并引领课程发展的趋势和方向）、示范性（教材所体现的课程体系具有较广泛的辐射性和示范性）和一定的前瞻

性。教材由个人申报或各校推荐(通过所在高校的“编委会”成员推荐),经“编委会”认真评审,最后由清华大学出版社审定出版。

目前,针对计算机类和电子信息类相关专业成立了两个“编委会”,即“清华大学出版社计算机教材编审委员会”和“清华大学出版社电子信息教材编审委员会”。首批推出的特色精品教材包括:

(1) 高等学校教材·计算机应用——高等学校各类专业,特别是非计算机专业的计算机应用类教材。

(2) 高等学校教材·计算机科学与技术——高等学校计算机相关专业的教材。

(3) 高等学校教材·电子信息——高等学校电子信息相关专业的教材。

(4) 高等学校教材·软件工程——高等学校软件工程相关专业的教材。

(5) 高等学校教材·信息管理与信息系统。

(6) 高等学校教材·财经管理与计算机应用。

清华大学出版社经过20多年的努力,在教材尤其是计算机和电子信息类专业教材出版方面树立了权威品牌,为我国的高等教育事业做出了重要贡献。清华版教材形成了技术准确、内容严谨的独特风格,这种风格将延续并反映在特色精品教材的建设中。

**清华大学出版社教材编审委员会**

**E-mail:dingl@tup.tsinghua.edu.cn**

# 前 言

作者多年讲授“数据结构”课程，所用教材为清华大学出版社出版，严蔚敏、吴伟民编著的《数据结构(C语言版)》(以下简称为教科书)。根据多年的授课经验，作者深知学习“数据结构”的关键点：

- 首先，要产生兴趣，兴趣是求知的动力。
- 其次，要加强形象思维训练，用形象思维帮助建立抽象思维。
- 最后，要使算法活起来，使算法不再是抽象的、枯燥的、孤立的、晦涩的，而是具体的、生动的、互相有联系的、易于理解的。

本书是作者多年来潜心研究的成果，其中有许多独到之处：

一、本书不仅包括教科书中绝大多数算法的实现，对于许多主要的存储结构，也包括了它们的基本操作的实现。这些基本操作构成了存储结构的完整体系，使得该存储结构可以直接使用在需要的地方。如在第7章的拓扑排序中就用到了第3章顺序栈的存储结构和基本操作。作者也经常直接将本书的存储结构和基本操作用在自己的科研课题程序中，效果都很好。读者如果需要了解少数教科书中提及而本书中未提及的算法、存储结构和基本操作，可以参考阅读本书后面的参考文献[3]。

二、为了加强形象思维训练，作者绘制了各种数据存储结构、算法、程序运行过程的示意图，共计281幅(有些图本身又由一系列小图组成)。这些图清楚地说明了数据的存储结构和算法。

三、通过将算法编写到计算机可运行的程序中的方法，使算法活起来。对于可运行的算法，输出变量、单步执行、设置断点、修改算法、尝试各种输入数据等都是轻而易举的，这些做法都有助于深刻地理解算法。

四、对于较难理解的算法都有详细的、图文并茂的解析，有些解析(如平衡二叉树)还包含作者自己的研究。较为简单的算法，也尽量利用程序中的空白处，多加注释。对于相应于教科书算法中须说明的新增行与修改行，在注释行中也分别注以“新增”与“修改”。

五、本书第7章以后的许多程序中的数据来自文本格式的数据文件，避免了人工键盘输入的麻烦，也有利于掌握使用文件输入输出的方法(这是很多学生不熟悉，却又很重要的方法)。根据程序所用文件的格式，读者很容易编写出自己需要的数据文件。

六、本书除了实现教科书中已有的算法，还实现了克鲁斯卡尔、2-路插入排序(包

括改进的 2-路插入排序)、树形选择排序等教科书中没有写出的算法。

七、本书还包含了许多编程的技巧和小窍门,这些是作者多年编程所积累的经验。

有教科书和本书对算法和数据结构的详细讲解,又有可执行的程序,还有程序的运行结果,加上读者自己的思考和努力,还有什么学不会的呢?甚至会觉得,数据结构是简单的,又是有趣的,还是有用的。其中的许多算法是巧妙的、启迪心智的,徜徉其中,其乐无穷。

本书紧密配合教科书,故在章节编排上尽量与教科书保持一致,以便读者对照查找。教科书的第 8 章和第 12 章由于内容较为独立,并应用较少,故没有收进本书。因此,教科书的第 9、10、11 章在本书中相应地成为第 8、9、10 章。同样,教科书中有些节的内容没有收进本书,后面节的编号随之减小。但各章节的名称与教科书保持一致。

本书所有程序都在 Borland C++ 3.1 和 Microsoft Visual C++ 6.0 下运行通过。这些程序都可通过清华大学出版社的网站(www.tup.tsinghua.edu.cn 或 www.tup.com.cn)下载。

本书虽然是以严蔚敏、吴伟民编著的《数据结构》(C 语言版)为基础,但对其他的数据结构教材的理解,也极具参考价值。

尽管作者尽了最大努力,但限于水平,书中疏漏之处在所难免,希望读者不吝赐教,以便将来改正。读者可通过本书的作者(gyfan@chd.edu.cn)与编辑(zhengyk@tup.tsinghua.edu.cn)取得联系。

作　者

2007 年 11 月于长安大学

# 目 录

# 第1章

# 绪　　论

本书内容主要由实现基本操作和算法的程序构成。这些程序文件有6类：

(1) 数据存储结构。文件名第一个字母为c，以h为扩展名。如c1-1.h是第1章的第1种存储结构。

(2) 每种存储结构的一组基本操作函数。以bo(bo表示基本操作)开头，cpp为扩展名。如bo1-1.cpp是第1章第1种存储结构的一组基本操作函数。教科书中涉及基本操作的算法也收到基本操作函数中，且在函数中注明算法的编号。

(3) 调用基本操作的主程序。以main(main表示主程序)开头，cpp为扩展名。如main1-1.cpp是调用bo1-1.cpp的主程序。

(4) 实现算法的程序。以algo(algo表示算法)开头，cpp为扩展名。如algo2-1.cpp是实现算法2.7的程序。

(5) 不属于基本操作又被多次调用的函数或程序段。以func开头，cpp为扩展名。如func2-2.cpp中的函数equal()、comp()、print()等被许多程序调用。为节约篇幅，将这些函数放到func2-2.cpp中。

(6) 数据文件。以txt为扩展名，如第7章的f7-1.txt等。

只有以main、algo开头的程序文件有主函数main()，而且是可执行的程序。其他程序都是被调用的程序，通过#include命令包括在可执行程序中，它们可能被多次调用，为了节省篇幅，只出现在书中第1次调用前。

## 1.1　抽象数据类型的表示与实现

教科书定义OK、ERROR等为函数的结果状态代码，Status为其类型，把这些信息放到头文件c1.h中。c1.h还包含一些常用的头文件，如string.h、stdio.h等。为了操作方便，本书几乎每一个程序都把c1.h包含进去，也就把这些结果状态代码的定义和头文件包含了进去。对于某一个程序来说，有些结果状态代码和头文件并没有用到，不过这不影响使用。头文件c1.h内容如下：

```
// c1.h (文件名)
#include<string.h> // 字符串函数头文件
#include<ctype.h> // 字符函数头文件
```

```
#include<malloc.h> // malloc()等
#include<limits.h> // INT_MAX 等
#include<stdio.h> // 标准输入输出头文件,包括 EOF( = ^Z 或 F6),NULL 等
#include<stdlib.h> // atoi(),exit()
#include<io.h> // eof()
#include<math.h> // 数学函数头文件,包括 floor(),ceil(),abs()等
#include<sys/timeb.h> // ftime()
#include<stdarg.h> // 提供宏 va_start,va_arg 和 va_end,用于存取变长参数表
// 函数结果状态代码。在教科书第 10 页
#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
// #define INFEASIBLE -1 没使用
// #define OVERFLOW -2 因为在 math.h 中已定义 OVERFLOW 的值为 3,故去掉此行
typedef int Status; // Status 是函数的类型,其值是函数结果状态代码,如 OK 等
typedef int Boolean; // Boolean 是布尔类型,其值是 TRUE 或 FALSE,第 7、8 章用到
```

教科书中的例 1-7 定义了一个三元组的抽象数据类型 Triplet。通过例 1-7,给出了这个三元组的表示方法和所定义的基本操作函数的实现。

例 1-7 实际上是教科书第 2～7 章中各种存储结构的一个范例:定义一种存储结构及建立在这种存储结构上的一组基本操作,并给出基本操作的实现。下面就是例 1-7 在程序中的具体实现。

头文件 c1-1.h 定义了三元组的抽象数据类型 Triplet。它采用动态分配的顺序存储结构(见图 1-1)。

Triplet ElemType

图 1-1 采用动态分配的顺序存储结构

```
// c1-1.h 采用动态分配的顺序存储结构。在教科书第 12 页
typedef ElemType * Triplet; // 由 InitTriplet 分配 3 个元素存储空间
// Triplet 类型是 ElemType 类型的指针,存放 ElemType 类型的地址
```

在 c1-1.h 中遇到 ElemType(元素类型),在后面的章节中还会遇到 SElemType(栈元素类型)、QElemType(队列元素类型)、TElemType(树元素类型)和 VertexType(图的顶点元素类型)等。在诸如 c1-1.h 这类头文件中,它们是抽象的数据类型,也称为多形数据类型。可以根据需要在主程中设置为具体的类型。

bo1-1.cpp 是有关抽象数据类型 Triplet 和 ElemType 的 8 个基本操作函数。这 8 个函数返回值的类型都是 Status,即函数返回值只能是头文件 c1.h 中定义的 OK、ERROR 等。

```
// bo1-1.cpp 抽象数据类型 Triplet 和 ElemType(由 c1-1.h 定义)的基本操作(8 个)
Status InitTriplet(Triplet &T,ElemType v1,ElemType v2,ElemType v3)
{ // 操作结果:构造三元组 T,依次置 T 的 3 个元素的初值为 v1,v2 和 v3。在教科书第 12 页(见图 1-2)
  T = (ElemType * )malloc(3 * sizeof(ElemType)); // 分配 3 个元素的存储空间
  if(!T)
    exit(OVERFLOW); // 分配失败则退出
  T[0] = v1,T[1] = v2,T[2] = v3;
  return OK;
```

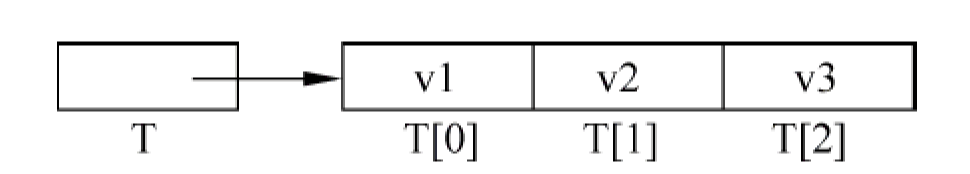



图 1-2 构造三元组 T

```
}
Status DestroyTriplet(Triplet &T)
{ // 操作结果：三元组 T 被销毁。在教科书第 12 页(见图 1-3)
  free(T); // 释放 T 所指的三元组存储空间
  T = NULL; // T 不再指向任何存储单元
  return OK;
}
```

| NULL |
|---|

T

图 1-3　三元组 T 被销毁

```
Status Get(Triplet T,int i,ElemType &e)
{ // 初始条件：三元组 T 已存在,1≤i≤3。操作结果：用 e 返回 T 的第 i 元的值。在教科书第 12 页
  if(i<1||i>3) // i 不在三元组的范围之内
    return ERROR;
  e = T[i - 1]; // 将三元组 T 的第 i 个元素的值赋给 e
  return OK;
}
Status Put(Triplet T,int i,ElemType e)
{ // 初始条件：三元组 T 已存在,1≤i≤3。操作结果：改变 T 的第 i 元的值为 e。在教科书第 12 页
  if(i<1||i>3) // i 不在三元组的范围之内
    return ERROR;
  T[i - 1] = e; // 将 e 的值赋给三元组 T 的第 i 个元素
  return OK;
}
Status IsAscending(Triplet T) // 在教科书第 13 页
{ // 初始条件：三元组 T 已存在。操作结果：如果 T 的 3 个元素按升序排列,则返回 1；否则返回 0
  return(T[0]<= T[1]&&T[1]<= T[2]); // 只在 T[0]不大于 T[1]且 T[1]不大于 T[2]时返回真
}
Status IsDescending(Triplet T) // 在教科书第 13 页
{ // 初始条件：三元组 T 已存在。操作结果：如果 T 的 3 个元素按降序排列,则返回 1；否则返回 0
  return(T[0]>= T[1]&&T[1]>= T[2]); // 只在 T[0]不小于 T[1]且 T[1]不小于 T[2]时返回真
}
Status Max(Triplet T,ElemType &e)
{ // 初始条件：三元组 T 已存在。操作结果：用 e 返回指向 T 的最大元素的值。在教科书第 13 页
  e = (T[0]>= T[1])?(T[0]>= T[2]?T[0]:T[2]):(T[1]>= T[2]?T[1]:T[2]); // 嵌套的条件运算符
  return OK;
}
Status Min(Triplet T,ElemType &e)
{ // 初始条件：三元组 T 已存在。操作结果：用 e 返回指向 T 的最小元素的值。在教科书第 13 页
  e = (T[0]<= T[1])?(T[0]<= T[2]?T[0]:T[2]):(T[1]<= T[2]?T[1]:T[2]); // 嵌套的条件运算符
  return OK;
}
```

main1-1. cpp 是检验 bo1-1. cpp 中的各基本操作函数是否正确的主函数。在 main1-1. cpp 中可根据需要定义抽象数据类型 ElemType 为 int、double 或其他数值型类型(只能是数值类型,因为其中有几个函数是比较大小的),而无须改变基本操作 bo1-1. cpp,这使得 bo1-1. cpp

的通用性大大增强。func1-1.cpp 是输出函数，根据 ElemType 在主函数中被定义的类型选择合适的语句。

```
// func1-1.cpp 输出函数
void PrintE(ElemType e) // 输出元素的值
{ printf("%d\n",e); // 定义 ElemType 为整型选此句。第 3 行
//printf("%f\n",e); // 定义 ElemType 为双精度型选此句。第 4 行
}
void PrintT(Triplet T) // 依次输出三元组的值
{ printf("%d,%d,%d\n",T[0],T[1],T[2]); // 定义 ElemType 为整型选此句。第 7 行
//printf("%f,%f,%f\n",T[0],T[1],T[2]); // 定义 ElemType 为双精度型选此句。第 8 行
}
```

```
// main1-1.cpp 检验基本操作 bo1-1.cpp 的主函数
#include"c1.h" // 要将程序中所有 #include 命令所包含的文件复制到当前目录下
// 以下两行可根据需要选其一(且只能选其一)，而无须改变基本操作 bo1-1.cpp
typedef int ElemType; // 定义抽象数据类型 ElemType 在本程序中为整型。第 4 行
//typedef double ElemType; // 定义抽象数据类型 ElemType 在本程序中为双精度型。第 5 行
#include"c1-1.h" // 在此命令之前要定义 ElemType 的类型
#include"bo1-1.cpp" // 在此命令之前要包括 c1-1.h 文件(因为其中定义了 Triplet)
#include"func1-1.cpp" // 输出函数，根据 ElemType 的类型选择不同的语句
void main()
{
  Triplet T;
  ElemType m;
  Status i;
  i=InitTriplet(T,5,7,9); // 初始化三元组 T，其 3 个元素依次为 5,7,9。第 14 行
//i=InitTriplet(T,5.0,7.1,9.3); // 当 ElemType 为双精度型时，可取代上句。第 15 行
  printf("调用初始化函数后，i=%d(1:成功)。T 的 3 个值为",i);
  PrintT(T); // 输出 T 的 3 个值
  i=Get(T,2,m); // 将三元组 T 的第 2 个值赋给 m
  if(i==OK) // 调用 Get()成功
  { printf("T 的第 2 个值为");
    PrintE(m); // 输出 m(=T[1])
  }
  i=Put(T,2,6); // 将三元组 T 的第 2 个值改为 6
  if(i==OK) // 调用 Put()成功
  { printf("将 T 的第 2 个值改为 6 后，T 的 3 个值为");
    PrintT(T); // 输出 T 的 3 个值
  }
  i=IsAscending(T); // 测试升序的函数
  printf("调用测试升序的函数后，i=%d(0:否 1:是)\n",i);
  i=IsDescending(T); // 测试降序的函数
  printf("调用测试降序的函数后，i=%d(0:否 1:是)\n",i);
```

```
  if((i = Max(T,m)) == OK) // 先赋值再比较
  { printf("T 中的最大值为");
    PrintE(m); // 输出最大值 m
  }
  if((i = Min(T,m)) == OK)
  { printf("T 中的最小值为");
    PrintE(m); // 输出最小值 m
  }
  DestroyTriplet(T); // 函数也可以不带回返回值
  printf("销毁 T 后,T = %u\n",T);
}
```

**程序运行结果：**

```
调用初始化函数后,i = 1(1:成功)。T 的 3 个值为 5,7,9
T 的第 2 个值为 7
将 T 的第 2 个值改为 6 后,T 的 3 个值为 5,6,9
调用测试升序的函数后,i = 1(0:否 1:是)
调用测试降序的函数后,i = 0(0:否 1:是)
T 中的最大值为 9
T 中的最小值为 5
销毁 T 后,T = 0
```

把 main1-1. cpp 的第 4、14 行注释掉,启用第 5、15 行,定义 ElemType 为 double 类型。同时把 func1-1. cpp 的第 3、7 行的语句注释掉(“{”不能注释掉),启用第 4、8 行,定义 ElemType 为 double 类型的输出语句,则程序运行结果如下：

```
调用初始化函数后,i = 1(1:成功)。T 的 3 个值为 5.000000,7.100000,9.300000
T 的第 2 个值为 7.100000
将 T 的第 2 个值改为 6 后,T 的 3 个值为 5.000000,6.000000,9.300000
调用测试升序的函数后,i = 1(0:否 1:是)
调用测试降序的函数后,i = 0(0:否 1:是)
T 中的最大值为 9.300000
T 中的最小值为 5.000000
销毁 T 后,T = 0
```

main1-1. cpp 是运行的第 1 个程序。通过运行 main1-1. cpp,要掌握这样几点：第一,main1-1. cpp 是提供本书程序风格的一个例子,它是教科书中例 1-7 的完整实现,通过它,把数据的存储结构、建立于此结构上的基本操作函数以及调用这些函数的主程序结合到一起；第二,将抽象的数据类型根据实际需要具体化的方法；第三,函数类型 Status 的应用：若函数类型为 Status,它的返回值只能是 OK、ERROR 等；第四,熟悉 C 语言中 malloc 函数的使用,在学习 C 语言时,malloc 函数使用得并不多,但在“数据结构”中它却几乎是使用最多的函数。

**注意：**只有将 c1. h、c1-1. h、bo1-1. cpp 和 func1-1. cpp 四个文件复制到 main1-1. cpp 的目录下,才能正确运行 main1-1. cpp。

本书的所有程序可以在 Borland C++ 3.1 环境下运行，也可在 Microsoft Visual C++6.0 环境下运行。本书中所列的程序运行结果是 Borland C++ 3.1 环境下的，它和 Microsoft Visual C++ 6.0 环境下的基本一样，只是整型最大值、指针变量的值等有所不同。

在 Microsoft Visual C++6.0 环境下运行的一种方法是：将所有需要的文件复制在同一个子目录下，再在“Windows 资源管理器”中双击含有主函数 main()的程序，进入到 VC++6.0 的环境。按 F7 键编译，没错误的话按 Ctrl+F5 键运行。另一种方法是：先进入 VC++6.0 的环境，单击主菜单 File，选 Open，在“打开”文件对话框中选择含有主函数 main()的程序，进入到 VC++6.0 的环境。仍然按 F7 键编译，没错误的话按 Ctrl+F5 键运行。如果是新编程序，单击主菜单 File 之后，选择 New 选项，在 Files 选项卡中选择 C++ Source File，给出 File name 和 Location(文件名和路径)后，单击 OK 按钮就可编写程序。

本书中的 main 主程序是用于检验相应 bo 程序中的基本操作各个函数程序是否正确的，它往往很长。若读者对某个基本操作函数特别关注，就可以对相应的 main 主程序进行删改，以适应自己的需要，又不必花太多的时间去输入程序。

在 bo1-1.cpp 中，有些基本函数的形参带有“&”，如第一个基本函数的声明：

```
Status InitTriplet(Triplet &T,ElemType v1,ElemType v2,ElemType v3);
```

其中，形参 T 前带有 &，说明形参 T 是引用类型的。引用类型是 C++ 语言特有的。引用类型的变量，其值若在函数中发生变化，则变化的值会带回主调函数中。程序 algo1-1.cpp 说明了函数中引用类型变量和非引用类型变量的区别。

```
// algo1-1.cpp 变量的引用类型和非引用类型的区别
#include"c1.h"
void fa(int a) // 在函数中改变 a,将不会带回主调函数(主调函数中的 a 仍是原值)
{ a++;
  printf("在函数 fa 中：a=%d\n",a);
}
void fb(int &a) // 由于 a 为引用类型,在函数中改变 a,其值将带回主调函数
{ a++;
  printf("在函数 fb 中：a=%d\n",a);
}
void main()
{
  int n=1;
  printf("在主程中,调用函数 fa 之前：n=%d\n",n);
  fa(n);
  printf("在主程中,调用函数 fa 之后,调用函数 fb 之前：n=%d\n",n);
  fb(n);
  printf("在主程中,调用函数 fb 之后：n=%d\n",n);
}
```

**程序运行结果：**

```
在主程中,调用函数 fa 之前：n = 1
在函数 fa 中：a = 2
在主程中,调用函数 fa 之后,调用函数 fb 之前：n = 1
在函数 fb 中：a = 2
在主程中,调用函数 fb 之后：n = 2
```

标准 C 语言中没有引用类型，如果需要在标准 C 语言环境下运行本书中的程序，则需要对程序做少许修改、转换，其方法详见附录 A。读者也可以参考本书后面的参考文献[3]一书所附带的光盘。其中，\TC 子目录下的程序是在 Turbo C 2.0 下运行通过的，且与\BC 子目录下的可在 Borland C++ 3.1 和 Microsoft Visual C++ 6.0 环境下运行的程序是逐个语句对应的。

## 1.2 算法和算法分析

同样是计算 1－1/x＋1/(x＊x)…，algo1-2.cpp 的语句频度表达式为(1＋n)×n/2，它的时间复杂度 $T(n)=O(n^2)$；而 algo1-3.cpp 的语句频度表达式为 n，它的时间复杂度 $T(n)=O(n)$。从两个程序的运行结果可以看出：当输入数据一样时，计算结果是一样的，但运行时间的差别很大。在算法正确的前提下，应该选择算法效率高的。

```
// algo1-2.cpp 计算 1 - 1/x + 1/(x * x)…
#include"c1.h"
void main()
{
  timeb t1,t2;
  long t;
  double x,sum = 1,sum1;
  int i,j,n;
  printf("请输入 x n：");
  scanf("%lf%d",&x,&n);
  ftime(&t1); // 求得当前时间
  for(i = 1;i<= n;i++)
  { sum1 = 1;
    for(j = 1;j<= i;j++)
      sum1 = sum1 * (-1.0/x);
    sum += sum1;
  }
  ftime(&t2); // 求得当前时间
  t = (t2.time - t1.time) * 1000 + (t2.millitm - t1.millitm); // 计算时间差
  printf("sum = %lf,用时 %ld 毫秒\n",sum,t);
}
```

**程序运行结果**(其中用时与计算机的配置有关，带下划线部分，由键盘输入)：

```
请输入 x n：123 10000↙
sum = 0.991935,用时 5440 毫秒
```

```
// algo1-3.cpp 计算 1 - 1/x + 1/(x * x)…的更快捷的算法
#include"c1.h"
void main()
{
  timeb t1,t2;
  long t;
  double x,sum1 = 1,sum = 1;
  int i,n;
  printf("请输入 x n:");
  scanf("%lf %d",&x,&n);
  ftime(&t1); // 求得当前时间
  for(i = 1;i<= n;i ++ )
  { sum1 *= - 1.0/x;
    sum += sum1;
  }
  ftime(&t2); // 求得当前时间
  t = (t2.time - t1.time) * 1000 + (t2.millitm - t1.millitm); // 计算时间差
  printf("sum = %lf,用时 %ld 毫秒\n",sum,t);
}
```

**程序运行结果：**

```
请输入 x n：123 10000↙
sum = 0.991935,用时 0 毫秒
```

# 第 2 章

# 线　性　表

## 2.1　线性表的类型定义

线性表的基本操作共有 12 个。通过将基本操作有机地组合,还可以对线性表进行较复杂的处理。算法 2.1 和算法 2.2(在 func2-1.cpp 中)就是这样的例子。

注意到算法 2.1 和算法 2.2 的形参 La、Lb 和 Lc 的类型是 List(表),List 并不是稍后将要介绍的具体的线性表存储结构如 SqList 和 LinkList 等,它是抽象的线性表类型。算法 2.1 和算法 2.2 中所涉及的函数都是线性表的基本操作,如 GetElem()、ListLength()等,所涉及的变量类型也都是 C 语言的数据类型,不涉及具体的线性表存储结构。这样,算法 2.1 和算法 2.2 就可以应用到任何一种具体的线性表存储结构中,只要这种存储结构的基本操作函数 GetElem()、ListLength()等已编写即可。

```
// func2-1.cpp 算法 2.1 和算法 2.2
void Union(List &La,List Lb) // 算法 2.1
{ // 将所有在线性表 Lb 中但不在 La 中的数据元素插入到表 La 中(不改变表 Lb)
  ElemType e;
  int La_len,Lb_len;
  int i;
  La_len = ListLength(La); // 求线性表 La 的长度
  Lb_len = ListLength(Lb); // 求线性表 Lb 的长度
  for(i = 1;i<= Lb_len;i ++ ) // 从表 Lb 的第 1 个元素到最后 1 个元素
  { GetElem(Lb,i,e); // 取表 Lb 中第 i 个数据元素的值赋给 e
    if(!LocateElem(La,e,equal)) // 表 La 中不存在和 e 相同的元素
      ListInsert(La,++ La_len,e); // 在表 La 的最后插入元素 e
  }
}
void MergeList(List La,List Lb,List &Lc) // 算法 2.2
{ // 已知线性表 La 和 Lb 中的数据元素按值非递减排列。
  // 归并 La 和 Lb 得到新的线性表 Lc,Lc 的数据元素也按值非递减排列(不改变表 La 和表 Lb)
  int i = 1,j = 1,k = 0;
  int La_len,Lb_len;
  ElemType ai,bj;
  InitList(Lc); // 创建空表 Lc
```

```
  La_len = ListLength(La); // 求线性表 La 的长度
  Lb_len = ListLength(Lb); // 求线性表 Lb 的长度
  while(i<= La_len&&j<= Lb_len) // i、j 分别指示表 La 和表 Lb 中的元素序号
  { GetElem(La,i,ai); // 取表 La 中第 i 个数据元素的值赋给 ai
    GetElem(Lb,j,bj); // 取表 Lb 中第 j 个数据元素的值赋给 bj
    if(ai<= bj) // 表 La 的当前元素不大于表 Lb 的当前元素
    { ListInsert(Lc,++ k,ai); // 在表 Lc 的最后插入元素 ai
      ++ i; // i 指示表 La 中的下一个元素
    }
    else
    { ListInsert(Lc,++ k,bj); // 在表 Lc 的最后插入元素 bj
      ++ j; // j 指示表 Lb 中的下一个元素
    }
  } // 以下两个 while 循环只会有一个被执行
  while(i<= La_len) // 表 La 中还有元素未插入到表 Lc
  { GetElem(La,i ++,ai); // 取表 La 中第 i 个数据元素的值赋给 ai,i 指示表 La 中的下一个元素
    ListInsert(Lc,++ k,ai); // 在表 Lc 的最后插入元素 ai
  }
  while(j<= Lb_len) // 表 Lb 中还有元素未插入到表 Lc
  { GetElem(Lb,j ++,bj); // 取表 Lb 中第 j 个数据元素的值赋给 bj,j 指示表 Lb 中的下一个元素
    ListInsert(Lc,++ k,bj); // 在表 Lc 的最后插入元素 bj
  }
}
```

可分别采用 SqList 和 LinkList 两种线性表存储结构调用 func2-1. cpp 实现算法 2. 1 和算法 2. 2 的程序是 2. 3. 1 节的 algo2-2. cpp。

## 2.2 线性表的顺序表示和实现

顺序表存储结构容易实现随机存取线性表的第 i 个数据元素的操作,但在实现插入或删除操作时要移动大量数据元素。所以,它适用于数据相对稳定的线性表,如职工工资表、学生学籍表等。

```
// c2-1.h 线性表的动态分配顺序存储结构。在教科书第 22 页(见图 2-1)
#define LIST_INIT_SIZE 10 // 线性表存储空间的初始分配量
#define LIST_INCREMENT 2 // 线性表存储空间的分配增量
struct SqList
{ ElemType * elem; // 存储空间基址
  int length; // 当前长度
  int listsize; // 当前分配的存储容量(以 sizeof(ElemType)为单位)
};
```

在图 2-1 中,用一个始于 elem 的箭头表示 elem 是指针类型。用该箭头止于 ElemType 类型表示 elem 是 ElemType 类型的指针。因为此处并不是说明 ElemType 类型,故用虚线

表示。图 2-2 是根据 c2-1.h 定义的有 2 个数据元素(2,6)、12 个存储空间的顺序表。




图 2-1　动态分配顺序存储结构

图 2-2　有 2 个数据元素(2,6)、12 个存储空间的顺序线性表 L

bo2-1.cpp 是基于顺序表的基本操作。由于 C++ 函数可重载,故去掉 bo2-1.cpp 中算法 2.3 等函数名中表示存储类型的后缀_Sq。c2-1.h 不采用固定数组作为线性表的存储结构,而是采用动态分配的存储结构,这样可以合理地利用空间,使长表占用的存储空间多,短表占用的存储空间少。这些可通过 bo2-1.cpp 中基本操作函数 ListInsert()和图 2-6 清楚地看出。

```
// bo2-1.cpp 顺序存储的线性表(存储结构由 c2-1.h 定义)的基本操作(12 个),包括算法 2.3～算法 2.6
void InitList(SqList &L) // 算法 2.3
{ // 操作结果:构造一个空的顺序线性表 L(见图 2-3)
  L.elem = (ElemType * )malloc(LIST_INIT_SIZE * sizeof(ElemType));
  if(!L.elem) // 存储分配失败
    exit(OVERFLOW);
  L.length = 0; // 空表长度为 0
  L.listsize = LIST_INIT_SIZE; // 初始存储容量
}
void DestroyList(SqList &L)
{ // 初始条件:顺序线性表 L 已存在。操作结果:销毁顺序线性表 L(见图 2-4)
  free(L.elem); // 释放 L.elem 所指的存储空间
  L.elem=NULL; // L.elem 不再指向任何存储单元
  L.length=0;
  L.listsize=0;
}
void ClearList(SqList &L)
{ // 初始条件:顺序线性表 L 已存在。操作结果:将 L 重置为空表(见图 2-5)
  L.length = 0;
}
Status ListEmpty(SqList L)
{ // 初始条件:顺序线性表 L 已存在。
  // 操作结果:若 L 为空表,则返回 TRUE;否则返回 FALSE
  if(L.length == 0)
    return TRUE;
  else
    return FALSE;
}
```

图 2-3　构造一个空的顺序线性表 L

图 2-4　销毁顺序线性表 L

图 2-5　有 12 个存储空间的空表 L

```
int ListLength(SqList L)
{ // 初始条件：顺序线性表 L 已存在。操作结果：返回 L 中数据元素的个数
  return L.length;
}
Status GetElem(SqList L,int i,ElemType &e)
{ // 初始条件：顺序线性表 L 已存在,1≤i≤ListLength(L)
  // 操作结果：用 e 返回 L 中第 i 个数据元素的值
  if(i<1||i>L.length) // i 不在表 L 的范围之内
    return ERROR;
  e=*(L.elem+i-1); // 将表 L 的第 i 个元素的值赋给 e
  return OK;
}
int LocateElem(SqList L,ElemType e,Status(*compare)(ElemType,ElemType))
{ // 初始条件：顺序线性表 L 已存在,compare()是数据元素判定函数(满足为 1,否则为 0)
  // 操作结果：返回 L 中第 1 个与 e 满足关系 compare()的数据元素的位序。
  //           若这样的数据元素不存在,则返回值为 0。算法 2.6
  int i=1; // i 的初值为第 1 个元素的位序
  ElemType *p=L.elem; // p 的初值为第 1 个元素的存储位置
  while(i<=L.length&&!compare(*p++,e)) // i 未超出表的范围且未找到满足关系的数据元素
    ++i; // 继续向后找
  if(i<=L.length) // 找到满足关系的数据元素
    return i; // 返回其位序
  else // 未找到满足关系的数据元素
    return 0;
}
Status PriorElem(SqList L,ElemType cur_e,ElemType &pre_e)
{ // 初始条件：顺序线性表 L 已存在
  // 操作结果：若 cur_e 是 L 的数据元素,且不是第一个,则用 pre_e 返回它的前驱;
  //           否则操作失败,pre_e 无定义
  int i=2; // 从第 2 个元素开始
  ElemType *p=L.elem+1; // p 指向第 2 个元素
  while(i<=L.length&&*p!=cur_e) // i 未超出表的范围且未找到值为 cur_e 的元素
  { p++; // p 指向下一个元素
    i++; // 计数加 1
  }
  if(i>L.length) // 到表结束处还未找到值为 cur_e 的元素
    return ERROR; // 操作失败
  else // 找到值为 cur_e 的元素,并由 p 指向其
  { pre_e=*--p; // p 指向前一个元素(cur_e 的前驱),将所指元素的值赋给 pre_e
    return OK; // 操作成功
  }
}
Status NextElem(SqList L,ElemType cur_e,ElemType &next_e)
{ // 初始条件：顺序线性表 L 已存在
  // 操作结果：若 cur_e 是 L 的数据元素,且不是最后一个,则用 next_e 返回它的后继,
```

```
//              否则操作失败,next_e 无定义
  int i = 1; // 从第 1 个元素开始
  ElemType * p = L.elem; // p 指向第 1 个元素
  while(i<L.length&& * p! = cur_e) // i 未到表尾且未找到值为 cur_e 的元素
  { p++; // p 指向下一个元素
    i++; // 计数加 1
  }
  if(i == L.length) // 到表尾的前一个元素还未找到值为 cur_e 的元素
    return ERROR; // 操作失败
  else // 找到值为 cur_e 的元素,并由 p 指向其
  { next_e = * ++p; // p 指向下一个元素(cur_e 的后继),将所指元素的值赋给 next _e
    return OK; // 操作成功
  }
}
Status ListInsert(SqList &L,int i,ElemType e) // 算法 2.4
{ // 初始条件:顺序线性表 L 已存在,1≤i≤ListLength(L) + 1
  // 操作结果:在 L 中第 i 个位置之前插入新的数据元素 e,L 的长度加 1(见图 2-6)
```

(a) L调用函数之前的状态　(b) L增加存储容量、移动元素　(c) L调用函数之后的状态

图 2-6　调用 ListInsert()示例(i=7,e=11)

```
  ElemType * newbase, * q, * p;
  if(i<1||i>L.length + 1) // i 值不合法
    return ERROR;
  if(L.length == L.listsize) // 当前存储空间已满,增加分配,修改
  { newbase = (ElemType * )realloc(L.elem,(L.listsize + LIST_INCREMENT) * sizeof(ElemType));
  if(!newbase) // 存储分配失败
      exit(OVERFLOW);
    L.elem = newbase; // 新基址赋给 L.elem
    L.listsize += LIST_INCREMENT; // 增加存储容量
  }
  q = L.elem + i - 1; // q 为插入位置
  for(p = L.elem + L.length - 1;p>= q;-- p) // 插入位置及之后的元素右移(由表尾元素开始移)
    * (p + 1) = * p;
  * q = e; // 插入 e
  ++ L.length; // 表长增 1
```

```
    return OK;
}
Status ListDelete(SqList &L,int i,ElemType &e) // 算法 2.5
{ // 初始条件：顺序线性表 L 已存在,1≤i≤ListLength(L)
  // 操作结果：删除 L 的第 i 个数据元素,并用 e 返回其值,L 的长度减 1(见图 2-7)
```

(a) L调用函数之前的状态　(b) L移动元素　(c) L调用函数之后的状态

图 2-7　调用 ListDelete()示例(i=7)

```
  ElemType *p,*q;
  if(i<1||i>L.length) // i 值不合法
    return ERROR;
  p=L.elem+i-1; // p 为被删除元素的位置
  e=*p; // 被删除元素的值赋给 e
  q=L.elem+L.length-1; // q 为表尾元素的位置
  for(++p;p<=q;++p) // 被删除元素之后的元素左移(由被删除元素的后继元素开始移)
    *(p-1)=*p;
  L.length--; // 表长减 1
  return OK;
}
void ListTraverse(SqList L,void(*visit)(ElemType&))
{ // 初始条件：顺序线性表 L 已存在
  // 操作结果：依次对 L 的每个数据元素调用函数 visit()
  //           visit()的形参加'&',表明可通过调用 visit()改变元素的值
  ElemType *p=L.elem; // p 指向第 1 个元素
  int i;
  for(i=1;i<=L.length;i++) // 从表 L 的第 1 个元素到最后 1 个元素
    visit(*p++); // 对每个数据元素调用 visit()
  printf("\n");
}
```

在 bo2-1.cpp 中,基本操作函数 ListTraverse()还要调用 visit()函数,并将 visit()函数设为 ListTraverse()的形参。把函数 visit()作为形参的原因,是要在 ListTraverse()中根据情况调用不同的函数而不是一个固定的函数。从作为形参的 visit()函数得知,满足哪些条件的函数可以被 ListTraverse()函数调用。在函数类形参的声明中指定了 visit()的函数类型,也就是函数返回值的类型(void)。在声明中还指定了 visit()函数形参的个数(1 个)和

类型(ElemType 的引用类型)。所有满足条件的函数(返回值为 void 类型,有一个形参,且类型为 ElemType&),都可以作为 ListTraverse()函数的实参。

LocateElem()函数也要调用一类函数 compare()。由 LocateElem()函数的声明可以看出:要求这类函数具有 2 个形参,其类型均为 ElemType;函数的返回值为 Status 类型。不仅如此,根据 LocateElem()函数的说明(初始条件),当 compare()函数的 2 个形参满足给定条件时,返回值为 1,否则为 0。只有满足这种条件的函数才能作为替代 compare()函数的实参函数。

func2-2.cpp 是一些常用的函数。main2-1.cpp 是检验 bo2-1.cpp 中的各基本操作函数是否正确的主函数。其中,sq()函数满足 LocateElem()函数对函数类形参 compare()的要求,可以作为 LocateElem()的调用函数;dbl()函数满足 ListTraverse()函数对函数类形参 visit()的要求,可以作为 ListTraverse()的调用函数。func2-2.cpp 中,print1()函数只是向屏幕输出形参 c,并不改变形参 c,所以形参不需要是引用类型,但为了和 ListTraverse()函数的要求一致,其形参被定义为引用类型。ListTraverse()函数之所以要求 visit()的形参为引用类型,是因为另一个实参函数 dbl()是给形参的值加倍,且要将形参值的改变带回主调函数,故必须是引用类型。

```
// func2-2.cpp 几个常用的函数
Status equal(ElemType c1,ElemType c2)
{ // 判断是否相等的函数
  if(c1 == c2)
    return TRUE;
  else
    return FALSE;
}
int comp(ElemType a,ElemType b)
{ // 根据 a<、= 或>b,分别返回 - 1、0 或 1
  if(a == b)
    return 0;
  else
    return (a - b)/abs(a - b);
}
void print(ElemType c)
{ // 以十进制整型的格式输出元素的值
  printf("%d",c);
}
void print1(ElemType &c)
{ // 以十进制整型的格式输出元素的值(设 c 为引用类型)
  printf("%d",c);
}
void print2(ElemType c)
{ // 以字符型的格式输出元素的值
  printf("%c",c);
}
```

```
// main2-1.cpp 检验 bo2-1.cpp 的主程序
#include"c1.h"
typedef int ElemType; // 定义 ElemType 为整型
#include"c2-1.h" // 线性表的顺序存储结构
#include"bo2-1.cpp" // 线性表顺序存储结构的基本操作
#include"func2-2.cpp" // 包括 equal()、comp()、print()、print1()和 print2()函数
Status sq(ElemType c1,ElemType c2)
{ // 数据元素判定函数(平方关系),LocateElem()调用的函数
  if(c1 == c2 * c2)
    return TRUE;
  else
    return FALSE;
}
void dbl(ElemType &c)
{ // ListTraverse()调用的另一函数(元素值加倍)
  c * = 2;
}
void main()
{
  SqList L;
  ElemType e,e0;
  Status i;
  int j,k;
  InitList(L); // 初始化线性表 L
  printf("初始化 L 后,L.length = %d,L.listsize = %d,L.elem = %u\n",L.length,
  L.listsize,L.elem);
  for(j = 1;j<= 5;j ++ )
    i = ListInsert(L,1,j); // 在 L 的表头插入 j
  printf("在 L 的表头依次插入 1~5 后,*L.elem = ");
  for(j = 1;j<= 5;j ++ )
    printf("%d", * (L.elem + j - 1)); // 依次输出表 L 中的元素
  printf("\n 调用 ListTraverse()函数,依次输出表 L 中的元素:");
  ListTraverse(L,print1); // 依次对表 L 中的元素调用 print1()函数(输出元素的值)
  i = ListEmpty(L); // 检测表 L 是否空
  printf("L.length = %d(改变),L.listsize = %d(不变),",L.length,L.listsize);
  printf("L.elem = %u(不变),L 是否空? i = %d(1:是 0:否)\n",L.elem,i);
  ClearList(L); // 清空表 L
  i = ListEmpty(L); // 再次检测表 L 是否空
  printf("清空 L 后,L.length = %d,L.listsize = %d,",L.length,L.listsize);
  printf("L.elem = %u,L 是否空? i = %d(1:是 0:否)\n",L.elem,i);
  for(j = 1;j<= 10;j ++ )
    ListInsert(L,j,j); // 在 L 的表尾插入 j
  printf("在 L 的表尾依次插入 1~10 后,L = ");
  ListTraverse(L,print1); // 依次输出表 L 中的元素
  printf("L.length = %d,L.listsize = %d,L.elem = %u\n",L.length,L.listsize,L.elem);
```

```
ListInsert(L,1,0); // 在 L 的表头插入 0,增加存储空间
printf("在 L 的表头插入 0 后,L.length = %d(改变),L.listsize = %d(改变),"
"L.elem = %u(有可能改变)\n",L.length,L.listsize,L.elem);
GetElem(L,5,e); // 将表 L 中的第 5 个元素的值赋给 e
printf("第 5 个元素的值为 %d\n",e);
for(j = 10;j<= 11;j ++ )
{ k = LocateElem(L,j,equal); // 查找表 L 中与 j 相等的元素,并将其位序赋给 k
  if(k) // k 不为 0,表明有符合条件的元素
    printf("第 %d 个元素的值为 %d,",k,j);
  else // k 为 0,没有符合条件的元素
    printf("没有值为 %d 的元素\n",j);
}
for(j = 3;j<= 4;j ++ ) // 测试 2 个数据
{ k = LocateElem(L,j,sq); // 查找表 L 中与 j 的平方相等的元素,并将其位序赋给 k
  if(k) // k 不为 0,表明有符合条件的元素
    printf("第 %d 个元素的值为 %d 的平方,",k,j);
  else // k 为 0,没有符合条件的元素
    printf("没有值为 %d 的平方的元素\n",j);
}
for(j = 1;j<= 2;j ++ ) // 测试头 2 个数据
{ GetElem(L,j,e0); // 将表 L 中的第 j 个元素的值赋给 e0
  i = PriorElem(L,e0,e); // 求 e0 的前驱,如成功,将值赋给 e
  if(i == ERROR) // 操作失败
    printf("元素 %d 无前驱,",e0);
  else // 操作成功
    printf("元素 %d 的前驱为 %d\n",e0,e);
}
for(j = ListLength(L) - 1;j<= ListLength(L);j ++ ) // 最后 2 个数据
{ GetElem(L,j,e0); // 将表 L 中的第 j 个元素的值赋给 e0
  i = NextElem(L,e0,e); // 求 e0 的后继,如成功,将值赋给 e
  if(i == ERROR) // 操作失败
    printf("元素 %d 无后继\n",e0);
  else // 操作成功
    printf("元素 %d 的后继为 %d,",e0,e);
}
k = ListLength(L); // k 为表长
for(j = k + 1;j>= k;j -- )
{ i = ListDelete(L,j,e); // 删除第 j 个数据
  if(i == ERROR) // 表中不存在第 j 个数据
    printf("删除第 %d 个元素失败。",j);
  else // 表中存在第 j 个数据,删除成功,其值赋给 e
    printf("删除第 %d 个元素成功,其值为 %d",j,e);
}
ListTraverse(L,dbl); // 依次对元素调用 dbl(),元素值乘 2
printf("L 的元素值加倍后,L = ");
```

```
  ListTraverse(L,print1); // 依次输出表 L 中的元素
  DestroyList(L); // 销毁表 L
  printf("销毁 L 后,L.length=%d,L.listsize=%d,L.elem=%u\n",L.length,
  L.listsize,L.elem);
}
```

**程序运行结果**(其中指针变量的值在不同的编译环境下会不同):

```
初始化 L 后,L.length=0,L.listsize=10,L.elem=2668
在 L 的表头依次插入 1~5 后,*L.elem=5 4 3 2 1
调用 ListTraverse()函数,依次输出表 L 中的元素:5 4 3 2 1
L.length=5(改变),L.listsize=10(不变),L.elem=2668(不变),L 是否空? i=0(1:是 0:否)
清空 L 后,L.length=0,L.listsize=10,L.elem=2668,L 是否空? i=1(1:是 0:否)
在 L 的表尾依次插入 1~10 后,L=1 2 3 4 5 6 7 8 9 10
L.length=10,L.listsize=10,L.elem=2668
在 L 的表头插入 0 后,L.length=11(改变),L.listsize=12(改变),L.elem=2692(有可能改变)
第 5 个元素的值为 4
第 11 个元素的值为 10,没有值为 11 的元素
第 10 个元素的值为 3 的平方,没有值为 4 的平方的元素
元素 0 无前驱,元素 1 的前驱为 0
元素 9 的后继为 10,元素 10 无后继
删除第 12 个元素失败。删除第 11 个元素成功,其值为 10
L 的元素值加倍后,L=0 2 4 6 8 10 12 14 16 18
销毁 L 后,L.length=0,L.listsize=0,L.elem=0
```

algo2-1.cpp 是采用线性表的动态分配顺序存储结构(c2-1.h)调用算法 2.7 的程序,其中包含了 c2-1.h 存储结构文件和 bo2-1.cpp 基本操作函数文件。

```
// algo2-1.cpp 实现算法 2.7 的程序
#include"c1.h"
typedef int ElemType; // 定义 ElemType 为整型
#include"c2-1.h" // 线性表的顺序存储结构
#include"bo2-1.cpp" // 线性表顺序存储结构的基本操作
#include"func2-2.cpp" // 包括 equal()、comp()、print()、print1()和 print2()函数
void MergeList(SqList La,SqList Lb,SqList &Lc) // 算法 2.7
{ // 已知顺序线性表 La 和 Lb 的元素按值非递减排列。
  // 归并 La 和 Lb 得到新的顺序线性表 Lc,Lc 的元素也按值非递减排列(不改变表 La 和表 Lb)
  ElemType *pa,*pa_last,*pb,*pb_last,*pc;
  pa=La.elem; // pa 指向表 La 的第 1 个元素
  pb=Lb.elem; // pb 指向表 Lb 的第 1 个元素
  Lc.listsize=Lc.length=La.length+Lb.length; // 不用 InitList()创建空表 Lc
  pc=Lc.elem=(ElemType *)malloc(Lc.listsize*sizeof(ElemType)); // 分配所需空间
  if(!Lc.elem) // 存储分配失败
    exit(OVERFLOW);
  pa_last=La.elem+La.length-1; // pa_last 指向表 La 的最后 1 个元素
  pb_last=Lb.elem+Lb.length-1; // pb_last 指向表 Lb 的最后 1 个元素
```

```
  while(pa<=pa_last&&pb<=pb_last) // 表 La 和表 Lb 均有元素没有归并
  { // 归并
    if(*pa<=*pb) // 表 La 的当前元素不大于表 Lb 的当前元素
      *pc++=*pa++; // 将 pa 所指单元的值赋给 pc 所指单元后,pa 和 pc 分别 +1
    else
      *pc++=*pb++; // 将 pb 所指单元的值赋给 pc 所指单元后,pb 和 pc 分别 +1
  } // 以下两个 while 循环只会有一个被执行
  while(pa<=pa_last) // 表 Lb 中的元素全都归并
    *pc++=*pa++; // 插入 La 的剩余元素
  while(pb<=pb_last) // 表 La 中的元素全都归并
    *pc++=*pb++; // 插入 Lb 的剩余元素
}
void main()
{
  SqList La,Lb,Lc;
  int j;
  InitList(La); // 创建空表 La
  for(j=1;j<=5;j++) // 在表 La 中插入 5 个元素,依次为 1、2、3、4、5(见图 2-8)
    ListInsert(La,j,j);
  printf("La= "); // 输出表 La 的内容
  ListTraverse(La,print1);
  InitList(Lb); // 创建空表 Lb
  for(j=1;j<=5;j++) // 在表 Lb 中插入 5 个元素,依次为 2、4、6、8、10(见图 2-9)
    ListInsert(Lb,j,2*j);
  printf("Lb= "); // 输出表 Lb 的内容
  ListTraverse(Lb,print1);
  MergeList(La,Lb,Lc); // 由按非递减排列的表 La、Lb 得到按非递减排列的表 Lc
  printf("Lc= "); // 输出表 Lc 的内容(见图 2-10)
  ListTraverse(Lc,print1);
}
```

图 2-8 表 La

图 2-9 表 Lb

图 2-10 表 Lc

**程序运行结果:**

```
La= 1 2 3 4 5
Lb= 2 4 6 8 10
Lc= 1 2 2 3 4 4 5 6 8 10
```

# 2.3 线性表的链式表示和实现

和顺序表相比，链表存储结构在实现插入、删除的操作时，不需要移动大量数据元素(但不容易实现随机存取线性表的第 i 个数据元素的操作)。所以，链表适用于经常需要进行插入和删除操作的线性表，如飞机航班的乘客表等。

## 2.3.1 线性链表

```
// c2-2.h 线性表的单链表存储结构。在教科书第 28 页(见图 2-11)
struct LNode
{ ElemType data;
  LNode *next;
};
typedef LNode *LinkList; // 另一种定义 LinkList 的方法
```

LinkList LNode LNode
data next

图 2-11 线性表的单链表存储结构

图 2-12 是根据 c2-2.h 定义的带有头结点且具有 2 个结点(4,7)的线性链表的结构。bo2-2.cpp 是这种带有头结点的线性链表的基本操作。

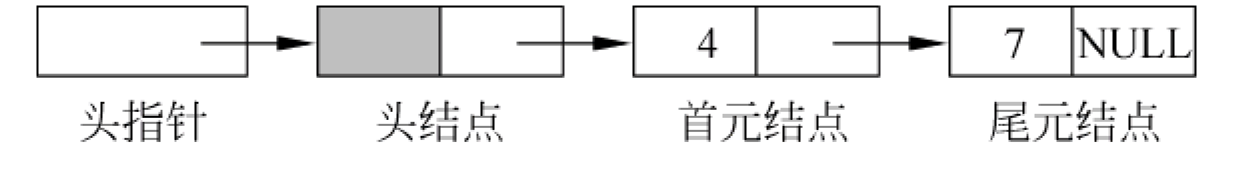



图 2-12 带有头结点且具有 2 个结点(4,7)的线性链表

```
// bo2-2.cpp 带有头结点的单链表(存储结构由 c2-2.h 定义)的基本操作(12 个),
// 包括算法 2.8~算法 2.10
void InitList(LinkList &L)
{ // 操作结果：构造一个空的线性表 L(见图 2-13)
  L=(LinkList)malloc(sizeof(LNode)); // 产生头结点,并使 L 指向此头结点
  if(!L) // 存储分配失败
    exit(OVERFLOW);
  L->next=NULL; // 头结点的指针域为空
}
```

NULL
L 头结点

图 2-13 空(仅有头结点)的单链表 L

```
void DestroyList(LinkList &L)
{ // 初始条件：线性表 L 已存在。操作结果：销毁线性表 L(见图 2-14)
  LinkList q;
  while(L) // L 指向结点(非空)
  { q=L->next; // q 指向首元结点
    free(L); // 释放头结点
    L=q; // L 指向原首元结点,现头结点
  }
}
```

NULL
L

图 2-14 线性表 L 被销毁

```
void ClearList(LinkList L) // 不改变 L
{ // 初始条件：线性表 L 已存在。操作结果：将 L 重置为空表(见图 2-13)
```

```
  LinkList p = L->next; // p 指向第 1 个结点
  L->next = NULL; // 头结点指针域为空
  DestroyList(p); // 销毁 p 所指的单链表
}
Status ListEmpty(LinkList L)
{ // 初始条件：线性表 L 已存在。操作结果：若 L 为空表，则返回 TRUE，否则返回 FALSE
  if(L->next) // 非空
    return FALSE;
  else
    return TRUE;
}
int ListLength(LinkList L)
{ // 初始条件：线性表 L 已存在。操作结果：返回 L 中数据元素的个数
  int i = 0; // 计数器初值为 0
  LinkList p = L->next; // p 指向第 1 个结点
  while(p) // 未到表尾
  { i++; // 计数器 + 1
    p = p->next; // p 指向下一个结点
  }
  return i;
}
Status GetElem(LinkList L,int i,ElemType &e) // 算法 2.8
{ // L 为带头结点的单链表的头指针。当第 i 个元素存在时，其值赋给 e 并返回 OK；否则返回 ERROR
  int j = 1; // 计数器初值为 1
  LinkList p = L->next; // p 指向第 1 个结点
  while(p&&j<i) // 顺指针向后查找，直到 p 指向第 i 个结点或 p 为空(第 i 个结点不存在)
  { j++; // 计数器 + 1
    p = p->next; // p 指向下一个结点
  }
  if(!p||j>i) // 第 i 个结点不存在
    return ERROR;
  e = p->data; // 取第 i 个元素的值赋给 e
  return OK;
}
int LocateElem(LinkList L,ElemType e,Status(* compare)(ElemType,ElemType))
{ // 初始条件：线性表 L 已存在，compare()是数据元素判定函数(满足为 1，否则为 0)
  // 操作结果：返回 L 中第 1 个与 e 满足关系 compare()的数据元素的位序。
  //           若这样的数据元素不存在，则返回值为 0
  int i = 0; // 计数器初值为 0
  LinkList p = L->next; // p 指向第 1 个结点
  while(p) // 未到表尾
  { i++; // 计数器 + 1
    if(compare(p->data,e)) // 找到这样的数据元素
      return i; // 返回其位序
    p = p->next; // p 指向下一个结点
```

```
    }
    return 0; // 满足关系的数据元素不存在
}
Status PriorElem(LinkList L,ElemType cur_e,ElemType &pre_e)
{ // 初始条件：线性表 L 已存在
  // 操作结果：若 cur_e 是 L 的数据元素，且不是第一个，则用 pre_e 返回它的前驱，返回 OK，
  //           否则操作失败，pre_e 无定义，返回 ERROR
  LinkList q,p = L->next; // p 指向第 1 个结点
  while(p->next) // p 所指结点有后继
  { q = p->next; // q 指向 p 的后继
    if(q->data == cur_e) // p 的后继为 cur_e
    { pre_e = p->data; // 将 p 所指元素的值赋给 pre_e
      return OK; // 成功返回 OK
    }
    p = q; // p 的后继不为 cur_e,p 向后移
  }
  return ERROR; // 操作失败，返回 ERROR
}
Status NextElem(LinkList L,ElemType cur_e,ElemType &next_e)
{ // 初始条件：线性表 L 已存在
  // 操作结果：若 cur_e 是 L 的数据元素，且不是最后一个，则用 next_e 返回它的后继，返回 OK，
  //           否则操作失败，next_e 无定义，返回 ERROR
  LinkList p = L->next; // p 指向第 1 个结点
  while(p->next) // p 所指结点有后继
  { if(p->data == cur_e) // p 所指结点的值为 cur_e
    { next_e = p->next->data; // 将 p 所指结点的后继结点的值赋给 next_e
      return OK; // 成功返回 OK
    }
    p = p->next; // p 指向下一个结点
  }
  return ERROR; // 操作失败，返回 ERROR
}
Status ListInsert(LinkList L,int i,ElemType e) // 算法 2.9 。不改变 L
{ // 在带头结点的单链线性表 L 中第 i 个位置之前插入元素 e(见图 2-15)
  int j = 0; // 计数器初值为 0
  LinkList s,p = L; // p 指向头结点
  while(p&&j<i-1) // 寻找第 i-1 个结点
  { j++; // 计数器+1
    p = p->next; // p 指向下一个结点
  }
  if(!p||j>i-1) // i 小于 1 或者大于表长
    return ERROR; // 插入失败
  s = (LinkList)malloc(sizeof(LNode)); // 生成新结点，以下将其插入 L 中
  s->data = e; // 将 e 赋给新结点
  s->next = p->next; // 新结点指向原第 i 个结点
```

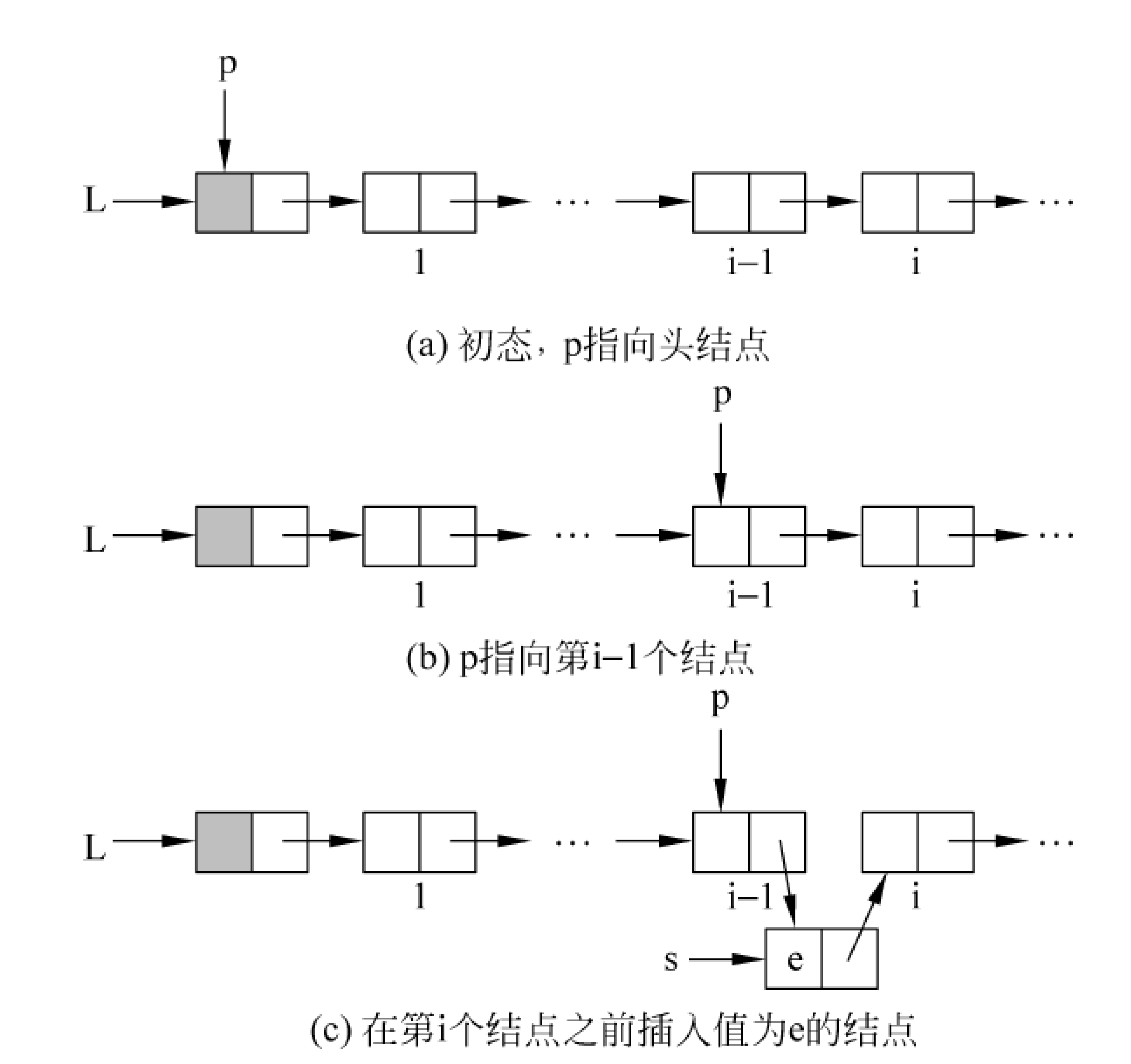



图 2-15　在链表 L 的第 i 个位置之前插入元素 e

```
  p->next = s; // 原第 i - 1 个结点指向新结点
  return OK; // 插入成功
}
Status ListDelete(LinkList L,int i,ElemType &e) // 算法 2.10 。不改变 L
{ // 在带头结点的单链线性表 L 中,删除第 i 个元素,并由 e 返回其值(见图 2-16)
```

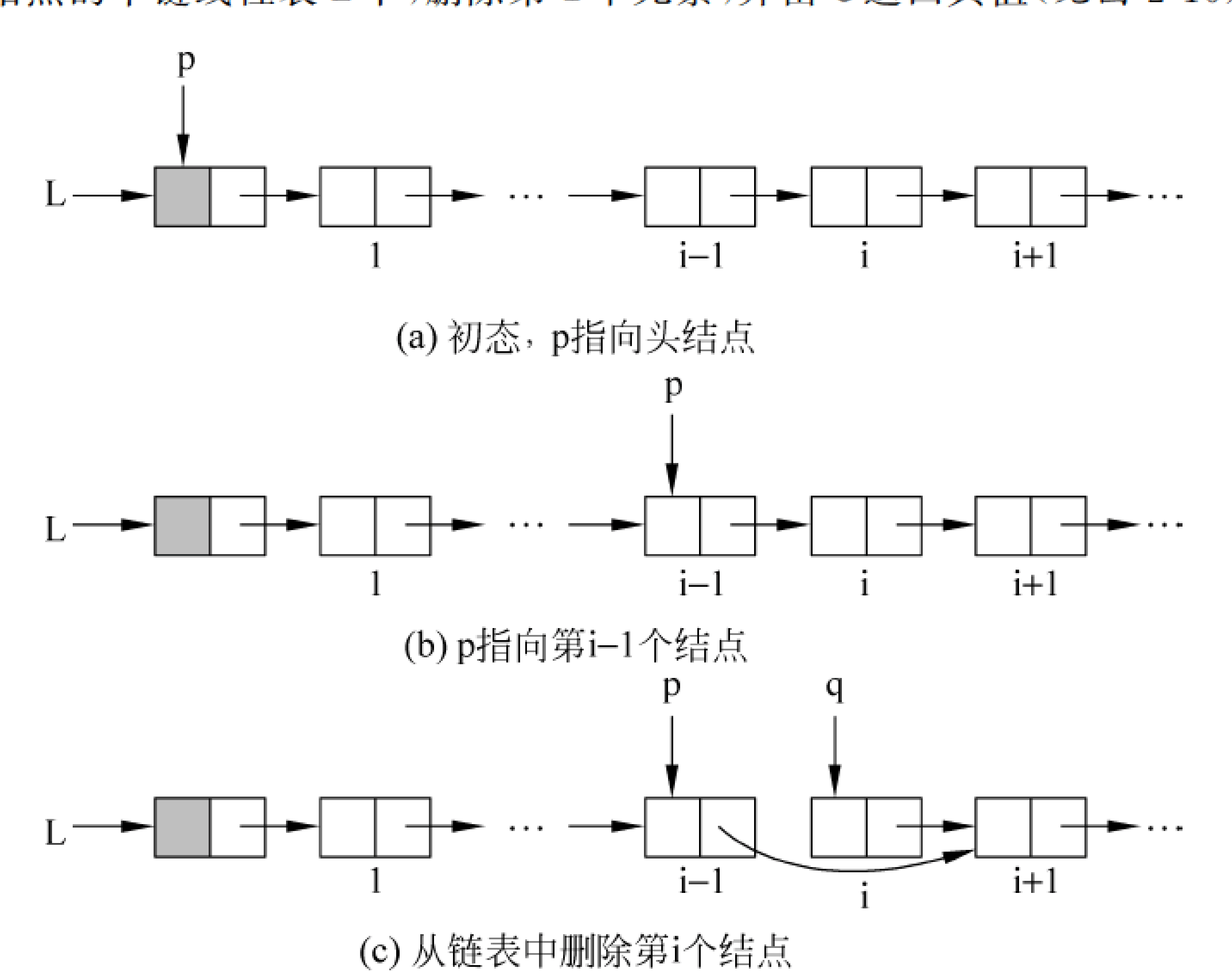



图 2-16　删除链表 L 的第 i 个结点

```
  int j = 0; // 计数器初值为 0
  LinkList q,p = L; // p 指向头结点
  while(p->next&&j<i - 1) // 寻找第 i 个结点,并令 p 指向其前驱
  { j ++; // 计数器 + 1
    p = p->next; // p 指向下一个结点
```

```
  }
  if(!p->next||j>i-1) // 删除位置不合理
    return ERROR; // 删除失败
  q=p->next; // q 指向待删除结点
  p->next=q->next; // 待删结点的前驱指向待删结点的后继
  e=q->data; // 将待删结点的值赋给 e
  free(q); // 释放待删结点
  return OK; // 删除成功
}
void ListTraverse(LinkList L,void(*visit)(ElemType))
// visit 的形参类型为 ElemType,与 bo2-1.cpp 中相应函数的形参类型 ElemType& 不同
{ // 初始条件:线性表 L 已存在。操作结果:依次对 L 的每个数据元素调用函数 visit()
  LinkList p=L->next; // p 指向第 1 个结点
  while(p) // p 所指结点存在
  { visit(p->data); // 对 p 所指结点调用函数 visit()
    p=p->next; // p 指向下一个结点
  }
  printf("\n");
}
```

main2-2.cpp 是验证基本操作 bo2-2.cpp 的主程序。为了后面的复用,主函数放在 func2-3.cpp 中。注意到 bo2-2.cpp 中的 ListTraverse()函数的形参 visit()函数的形参类型是 ElemType,不是引用类型 ElemType &,这与 bo2-1.cpp 中的 ListTraverse()函数不同。因此在主程序中替代 visit()的实参函数 print()的形参类型也是 ElemType。由于 bo2-2.cpp 和 bo2-1.cpp 都是 12 个函数名、操作结果均相同的基本操作函数,仅变量类型、实现过程不同,而验证基本操作的主程序 main2-1.cpp 和 main2-2.cpp 的作用又基本相同,所以程序 func2-3.cpp 和程序 main2-1.cpp 中的主函数很相像。

```
// func2-3.cpp 检验单链表基本操作的主函数
// main2-2.cpp、main2-3.cpp、main2-4.cpp 和 main2-5.cpp 调用
void main()
{
  LinkList L; // 与 main2-1.cpp 不同
  ElemType e,e0;
  Status i;
  int j,k;
  InitList(L); // 初始化线性表 L
  for(j=1;j<=5;j++)
    i=ListInsert(L,1,j); // 在 L 的表头插入 j
  printf("在 L 的表头依次插入 1~5 后,L=");
  ListTraverse(L,print); // 依次对元素调用 print(),输出元素的值
  i=ListEmpty(L); // 检测表 L 是否空
  printf("L 是否空? i=%d(1:是 0:否),表 L 的长度=%d\n",i,ListLength(L));
  ClearList(L); // 清空表 L
  printf("清空 L 后,L=");
```

```
    ListTraverse(L,print);
    i = ListEmpty(L); // 再次检测表 L 是否空
    printf("L 是否空? i = %d(1:是 0:否),表 L 的长度 = %d\n",i,ListLength(L));
    for(j = 1;j<= 10;j ++ )
      ListInsert(L,j,j); // 在 L 的表尾插入 j
    printf("在 L 的表尾依次插入 1~10 后,L = ");
    ListTraverse(L,print); // 依次输出表 L 中的元素
    for(j = 0;j<= 1;j ++ )
    {
  #ifdef SLL // 仅用于静态链表
      k = LocateElem(L,j); // 查找表 L 中与 j 相等的元素,并将其位序赋给 k
      if(k) // k 不为 0,表明有符合条件的元素
        printf("值为 %d 的元素的位序为 %d\n",j,k);
  #else // 仅用于链表
      k = LocateElem(L,j,equal); // 查找表 L 中与 j 相等的元素,并将其在链表中的排序赋给 k
      if(k) // k 不为 0,表明有符合条件的元素
        printf("第 %d 个元素的值为 %d\n",k,j);
  #endif
      else // k 为 0,没有符合条件的元素
        printf("没有值为 %d 的元素,",j);
    }
    for(j = 1;j<= 2;j ++ ) // 测试头 2 个数据
    { GetElem(L,j,e0); // 把表 L 中的第 j 个数据赋给 e0
      i = PriorElem(L,e0,e); // 求 e0 的前驱,如成功,将值赋给 e
      if(i == ERROR) // 操作失败
        printf("元素 %d 无前驱,",e0);
      else // 操作成功
        printf("元素 %d 的前驱为 %d\n",e0,e);
    }
    for(j = ListLength(L) - 1;j<= ListLength(L);j ++ ) // 最后 2 个数据
    { GetElem(L,j,e0); // 把表 L 中的第 j 个数据赋给 e0
      i = NextElem(L,e0,e); // 求 e0 的后继,如成功,将值赋给 e
      if(i == ERROR) // 操作失败
        printf("元素 %d 无后继\n",e0);
      else // 操作成功
        printf("元素 %d 的后继为 %d,",e0,e);
    }
    k = ListLength(L); // k 为表长
    for(j = k + 1;j>= k;j -- )
    { i = ListDelete(L,j,e); // 删除第 j 个数据
      if(i == ERROR) // 表中不存在第 j 个数据
        printf("删除第 %d 个元素失败(不存在此元素)。",j);
      else // 表中存在第 j 个数据,删除成功,其值赋给 e
        printf("删除第 %d 个元素成功,其值为 %d\n",j,e);
    }
```

```
  printf("依次输出 L 的元素：");
  ListTraverse(L,print); // 依次输出表 L 中的元素
  DestroyList(L); // 销毁表 L
#ifndef SLL // 仅用于链表
  printf("销毁 L 后,L=%u\n",L);
#endif
}
```

```
// main2-2.cpp 检验 bo2-2.cpp 的主程序
#include"c1.h"
typedef int ElemType; // 定义 ElemType 为整型
#include"c2-2.h" // 线性表的单链表存储结构
#include"bo2-2.cpp" // 设有头结点单链表存储结构的基本操作
#include"func2-2.cpp" // 包括 equal()、comp()、print()、print1()和 print2()函数
#include"func2-3.cpp" // 主函数
```

**程序运行结果：**

```
在 L 的表头依次插入 1～5 后,L=5 4 3 2 1
L 是否空？ i=0(1:是 0:否),表 L 的长度=5
清空 L 后,L=
L 是否空？ i=1(1:是 0:否),表 L 的长度=0
在 L 的表尾依次插入 1～10 后,L=1 2 3 4 5 6 7 8 9 10
没有值为 0 的元素,第 1 个元素的值为 1
元素 1 无前驱,元素 2 的前驱为 1
元素 9 的后继为 10,元素 10 无后继
删除第 11 个元素失败(不存在此元素)。删除第 10 个元素成功,其值为 10
依次输出 L 的元素：1 2 3 4 5 6 7 8 9
销毁 L 后,L=0
```

```
// algo2-2.cpp 用 SqList 类型和 LinkList 类型分别实现算法 2.1 和算法 2.2 的程序
#include"c1.h"
typedef int ElemType; // 定义 ElemType 为整型
#define Sq // (用 SqList 类型选此行,用 LinkList 类型将此行作为注释)
#ifdef Sq
  #include"c2-1.h" // 采用线性表的动态分配顺序存储结构
  #include"bo2-1.cpp" // 可以使用 bo2-1.cpp 中的基本操作
  typedef SqList List; // 定义抽象数据类型 List 为 SqList 类型
  #define printer print1 // ListTraverse()用到不同类型的输出函数
#else
  #include"c2-2.h" // 采用线性表的单链表存储结构
  #include"bo2-2.cpp" // 可以使用 bo2-2.cpp 中的基本操作
  typedef LinkList List; // 定义抽象数据类型 List 为 LinkList 类型
  #define printer print // ListTraverse()用到不同类型的输出函数
#endif
#include"func2-2.cpp" // 包括 equal()、comp()、print()、print1()和 print2()函数
```

```
#include"func2-1.cpp" // 包括算法 2.1 和算法 2.2
void main()
{
  List La,Lb,Lc;
  int j,b[7]={2,6,8,9,11,15,20};
  InitList(La); // 创建空表 La。如不成功,则会退出程序的运行
  for(j=1;j<=5;j++) // 在表 La 中插入 5 个元素,依次为 1、2、3、4、5
    ListInsert(La,j,j);
  printf("La= ");
  ListTraverse(La,printer); // 输出表 La 的内容
  InitList(Lb); // 创建空表 Lb
  for(j=1;j<=5;j++) // 在表 Lb 中插入 5 个元素,依次为 2、4、6、8、10
    ListInsert(Lb,j,2*j);
  printf("Lb= ");
  ListTraverse(Lb,printer); // 输出表 Lb 的内容
  Union(La,Lb); // 调用算法 2.1,将 Lb 中满足条件的元素插入 La(不改变 Lb)
  printf("new La= ");
  ListTraverse(La,printer); // 输出新表 La 的内容
  ClearList(Lb); // 清空表 Lb
  for(j=1;j<=7;j++) // 在表 Lb 中重新依次插入数组 b[]的 7 个元素
    ListInsert(Lb,j,b[j-1]);
  printf("Lb= ");
  ListTraverse(Lb,printer); // 输出表 Lb 的内容
  MergeList(La,Lb,Lc); // 调用算法 2.2,生成新表 Lc(不改变表 La 和表 Lb)
  printf("Lc= ");
  ListTraverse(Lc,printer); // 输出表 Lc 的内容
}
```

**程序运行结果:**

```
La= 1 2 3 4 5
Lb= 2 4 6 8 10
new La= 1 2 3 4 5 6 8 10
Lb= 2 6 8 9 11 15 20
Lc= 1 2 2 3 4 5 6 6 8 8 9 10 11 15 20
```

algo2-2.cpp 利用条件编译,通过是否定义了宏名 Sq,选取“typedef SqList List;”或“typedef LinkList List;”语句,将 func2-1.cpp 中的抽象数据类型 List 定义为 SqList 类型或 LinkList 类型,并且包含不同的存储结构文件和基于不同存储结构的基本操作函数文件。这样,就可以在线性表的顺序存储结构和链式存储结构中都能调用 func2-1.cpp 中的 Union()和 MergeList()函数了。

func2-1.cpp 中的 Union()和 MergeList()函数能用于不同存储结构的原因在于 Union()和 MergeList()函数是通过调用线性表的基本操作(如 ListLength()等)来完成的。算法中不包括针对某种具体的存储结构所特有的变量的操作。这类算法的可移植性好,从一种存储结构移植到另一种存储结构仅做少量修改即可。但由于没有考虑具体的存储结构,这类算

法往往不是最高效的。而算法 2.7 的程序(在 algo2-1.cpp 中)涉及顺序存储结构所特有的 L.length 等变量,不容易移植到其他存储结构中使用。但这类算法可以充分考虑存储结构的特点,从而做到高效。

教科书中有类似特性的还有算法 4.1 和算法 7.4~算法 7.8,它们都是基于抽象数据类型 String 或 Graph 编写的算法。

```
// algo2-3.cpp 实现算法 2.11 和算法 2.12 的程序
#include"c1.h"
typedef int ElemType; // 定义 ElemType 为整型
#include"c2-2.h" // 线性表的单链表存储结构
#include"bo2-2.cpp" // 设有头结点单链表存储结构的基本操作
#include"func2-2.cpp" // 包括 equal()、comp()、print()、print1()和 print2()函数
void CreateList(LinkList &L,int n) // 算法 2.11
{ // 逆位序(结点插在表头)输入 n 个元素的值,建立带表头结点的单链线性表 L
  int i;
  LinkList p;
  L=(LinkList)malloc(sizeof(LNode)); // 生成头结点
  L->next=NULL; // 先建立一个带头结点的空单链表
  printf("请输入 %d 个数据\n",n);
  for(i=n;i>0;--i)
  { p=(LinkList)malloc(sizeof(LNode)); // 生成新结点
    scanf("%d",&p->data); // 给新结点输入元素值
    p->next=L->next; // 将新结点插在表头
    L->next=p; // 头结点指向新结点
  }
}
void CreateList1(LinkList &L,int n)
{ // 正位序(结点插在表尾)输入 n 个元素的值,建立带表头结点的单链线性表 L
  int i;
  LinkList p,q;
  L=(LinkList)malloc(sizeof(LNode)); // 生成头结点
  L->next=NULL; // 先建立一个带头结点的空单链表
  q=L; // q 指向空表的头结点(相当于尾结点)
  printf("请输入 %d 个数据\n",n);
  for(i=1;i<=n;i++)
  { p=(LinkList)malloc(sizeof(LNode)); // 生成新结点
    scanf("%d",&p->data); // 给新结点输入元素值
    q->next=p; // 将新结点插在表尾
    q=q->next; // q 指向尾结点
  }
  p->next=NULL; // 最后一个结点的指针域为空
}
void MergeList(LinkList La,LinkList &Lb,LinkList &Lc) // 算法 2.12
{ // 已知单链线性表 La 和 Lb 的元素按值非递减排列。
```

```
  // 归并 La 和 Lb 得到新的单链线性表 Lc,Lc 的元素也按值非递减排列。(销毁 Lb,Lc 即新的 La)
  LinkList pa = La->next,pb = Lb->next,pc; // pa、pb 分别指向 La、Lb 的首元结点(待比较结点)
  Lc = pc = La; // 用 La 的头结点作为 Lc 的头结点,pc 指向 La 的头结点(Lc 的尾结点)
  while(pa&&pb) // La 和 Lb 中的元素都未比较完
    if(pa->data<= pb->data) // La 的当前元素不大于 Lb 的当前元素
    { pc->next = pa; // 将 pa 所指结点归并到 Lc 中
      pc = pa; // pc 指向表 Lc 的最后一个结点
      pa = pa->next; // 表 La 的下一个结点成为待比较结点
    }
    else // Lb 的当前元素小于 La 的当前元素
    { pc->next = pb; // 将 pb 所指结点归并到 Lc 中
      pc = pb; // pc 指向表 Lc 的最后一个结点
      pb = pb->next; // 表 Lb 的下一个结点成为待比较结点
    }
  pc->next = pa? pa:pb; // 插入剩余段
  free(Lb); // 释放 Lb 的头结点
  Lb = NULL; // Lb 不再指向任何结点
}
void main()
{
  int n = 5;
  LinkList La,Lb,Lc;
  printf("按非递减顺序,");
  CreateList1(La,n); // 根据输入顺序,正位序建立线性表
  printf("La = ");
  ListTraverse(La,print); // 输出链表 La 的内容
  printf("按非递增顺序,");
  CreateList(Lb,n); // 根据输入顺序,逆位序建立线性表
  printf("Lb = ");
  ListTraverse(Lb,print); // 输出链表 Lb 的内容
  MergeList(La,Lb,Lc); // 按非递减顺序归并 La 和 Lb,得到新表 Lc
  printf("Lc = ");
  ListTraverse(Lc,print); // 输出链表 Lc 的内容
}
```

**程序运行结果:**

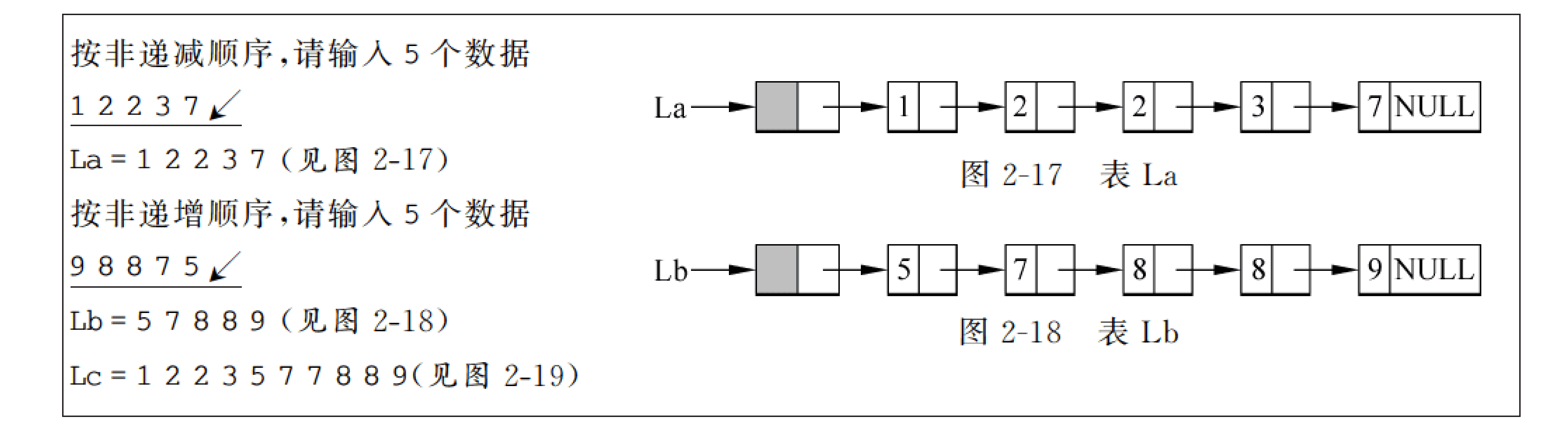


图 2-17　表 La

图 2-18　表 Lb

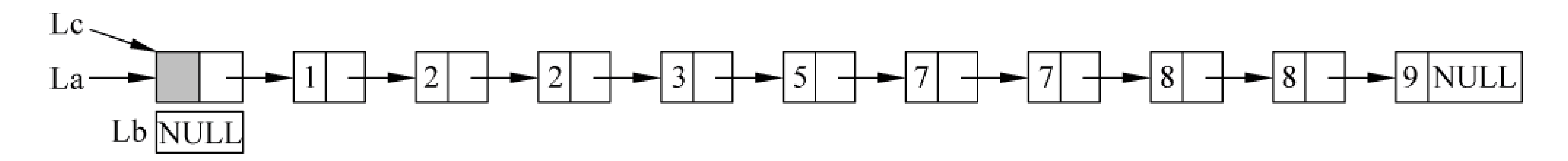



图 2-19　调用 MergeList()，归并表 La、Lb，得到表 Lc

单链表也可以不设头结点，如图 2-20 所示。显然，基于这种结构的基本操作和带有头结点的线性链表基本操作是不同的。bo2-3. cpp 和 bo2-4. cpp 是不设头结点的单链表的基本操作。

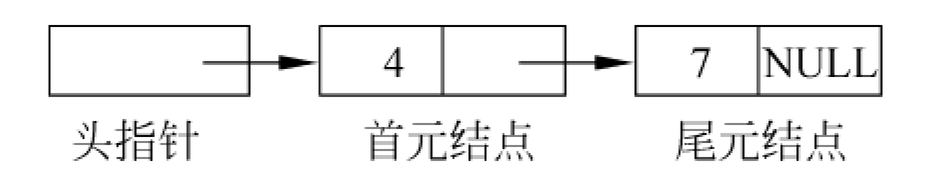



图 2-20　不设头结点且具有 2 个结点(4，7)的线性链表

```
// bo2-3.cpp 不设头结点的单链表(存储结构由 c2-2.h 定义)的部分基本操作(9 个)
#define DestroyList ClearList // DestroyList()和 ClearList()的操作是一样的
void InitList(LinkList &L)
{ // 操作结果：构造一个空的线性表 L(见图 2-21)
  L = NULL; // 指针为空
}
```

NULL

L

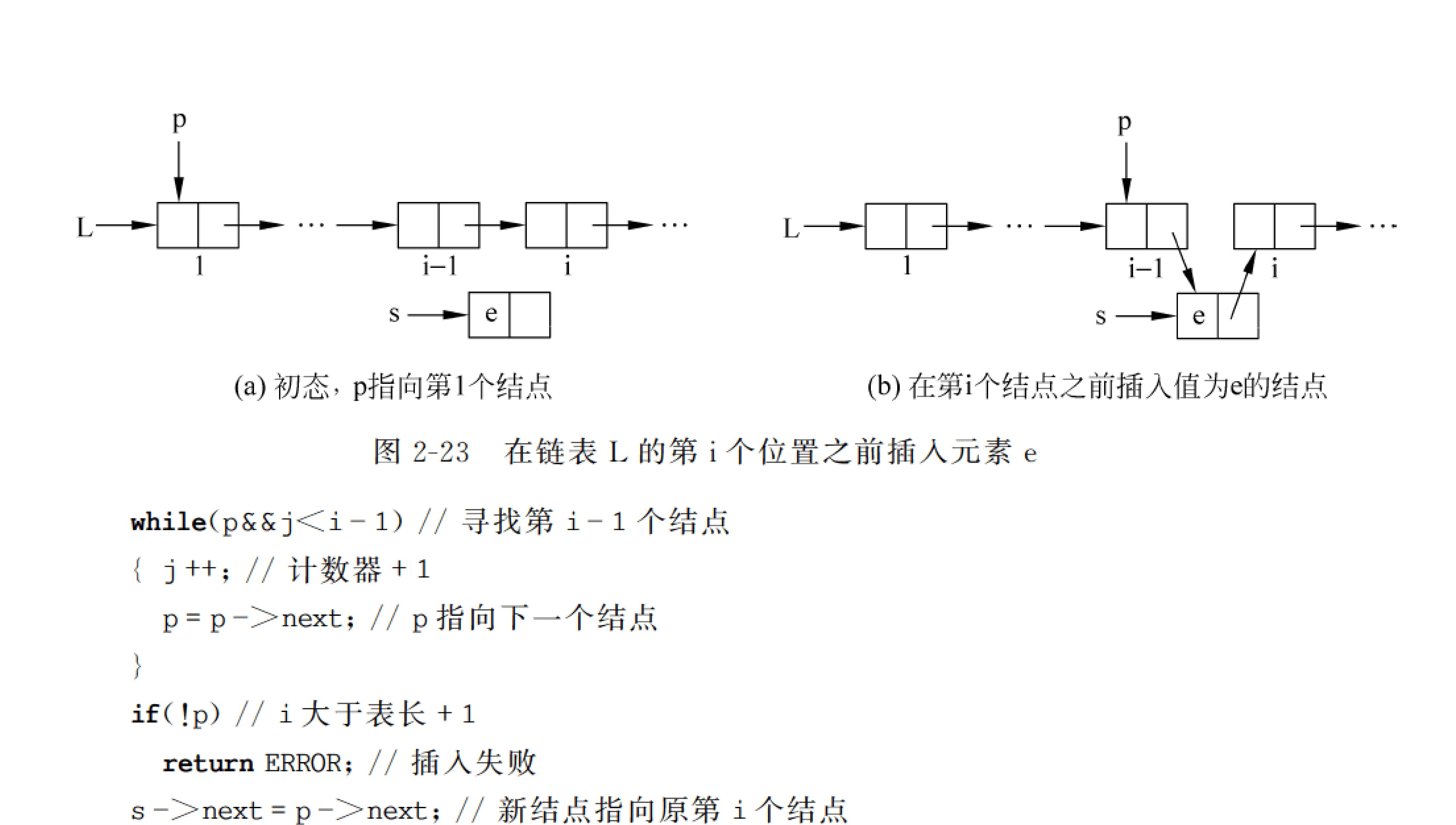
图 2-21　空的和被销毁的线性表 L

```
void ClearList(LinkList &L)
{ // 初始条件：线性表 L 已存在。操作结果：将 L 重置为空表(见图 2-21)
  LinkList p;
  while(L) // L 不空
  { p = L; // p 指向首元结点
    L = L->next; // L 指向第 2 个结点(新首元结点)
    free(p); // 释放首元结点
  }
}
Status ListEmpty(LinkList L)
{ // 初始条件：线性表 L 已存在。操作结果：若 L 为空表，则返回 TRUE，否则返回 FALSE
  if(L)
    return FALSE;
  else
    return TRUE;
}
int ListLength(LinkList L)
{ // 初始条件：线性表 L 已存在。操作结果：返回 L 中数据元素的个数
  int i = 0; // 计数器初值为 0
  LinkList p = L; // p 指向第 1 个结点
  while(p) // p 指向结点(未到表尾)
  { i++; // 计数器 + 1
    p = p->next; // p 指向下一个结点
  }
  return i;
}
```

```
Status GetElem(LinkList L,int i,ElemType &e)
{ // L 为不设头结点的单链表的头指针。当第 i 个元素存在时,其值赋给 e 并返回 OK,
 // 否则返回 ERROR
  int j = 1; // 计数器初值为 1
  LinkList p = L; // p 指向第 1 个结点
  if(i<1) // i 值不合法
    return ERROR;
  while(j<i&&p) // 未到第 i 个元素,也未到表尾
  { j ++; // 计数器 + 1
    p = p ->next; // p 指向下一个结点
  }
  if(j == i&&p) // 存在第 i 个元素
  { e = p ->data; // 取第 i 个元素的值赋给 e
    return OK; // 成功返回 OK
  }
  return ERROR; // 不存在第 i 个元素,失败返回 ERROR
}
int LocateElem(LinkList L,ElemType e,Status(* compare)(ElemType,ElemType))
{ // 初始条件:线性表 L 已存在,compare()是数据元素判定函数(满足为 1,否则为 0)
  // 操作结果:返回 L 中第 1 个与 e 满足关系 compare()的数据元素的位序。
  //          若这样的数据元素不存在,则返回值为 0
  int i = 0; // 计数器初值为 0
  LinkList p = L; // p 指向第 1 个结点
  while(p) // 未到表尾
  { i ++; // 计数器 + 1
    if(compare(p ->data,e)) // 找到这样的数据元素
      return i; // 返回其位序
    p = p ->next; // p 指向下一个结点
  }
  return 0; // 满足关系的数据元素不存在
}
Status ListInsert(LinkList &L,int i,ElemType e)
{ // 在不设头结点的单链线性表 L 中第 i 个位置之前插入元素 e
  int j = 1; // 计数器初值为 1
  LinkList s,p = L; // p 指向第 1 个结点
  if(i<1) // i 值不合法
    return ERROR;
  s = (LinkList)malloc(sizeof(LNode)); // 生成新结点,以下将其插入 L 中
  s ->data = e; // 给 s 的 data 域赋值 e
  if(i == 1) // 插在表头(见图 2-22)
  { s ->next = L; // 新结点指向原第 1 个结点
    L = s; // L 指向新结点(改变 L)
  }
  else
  { // 插在表的其余处(见图 2-23)
```

(a) 初态　　(b) 插在表头

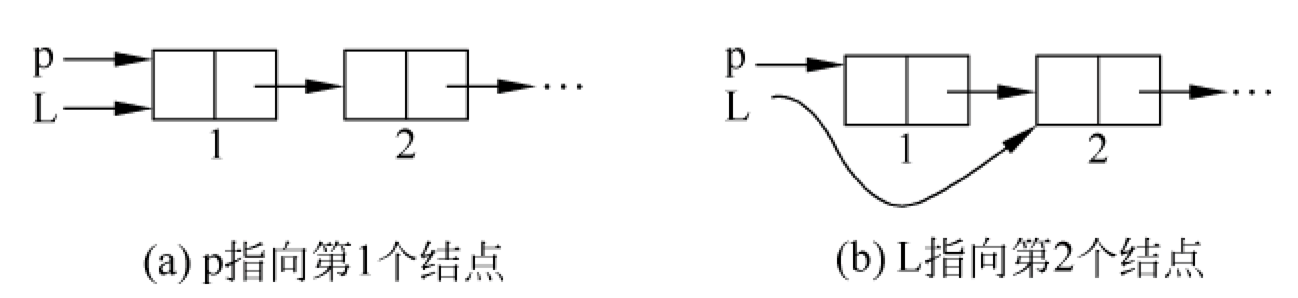
图 2-22　在链表 L 的第 1 个位置之前插入元素 e




(a) 初态，p指向第1个结点　　(b) 在第i个结点之前插入值为e的结点

图 2-23　在链表 L 的第 i 个位置之前插入元素 e

```
    while(p&&j<i-1) // 寻找第 i-1 个结点
    { j++; // 计数器+1
      p=p->next; // p 指向下一个结点
    }
    if(!p) // i 大于表长+1
      return ERROR; // 插入失败
    s->next=p->next; // 新结点指向原第 i 个结点
    p->next=s; // 原第 i-1 个结点指向新结点
  }
  return OK; // 插入成功
}
Status ListDelete(LinkList &L,int i,ElemType &e)
{ // 在不设头结点的单链线性表 L 中,删除第 i 个元素,并由 e 返回其值
  int j=1; // 计数器初值为 1
  LinkList q,p=L; // p 指向第 1 个结点
  if(!L) // 表 L 空
    return ERROR; // 删除失败
  else if(i==1) // 删除第 1 个结点(见图 2-24)
```




(a) p指向第1个结点　　(b) L指向第2个结点

图 2-24　删除链表 L 的第 1 个结点

```
  { L=p->next; // L 由第 2 个结点开始(改变 L)
    e=p->data; // 将待删结点的值赋给 e
    free(p); // 删除并释放第 1 个结点
  }
  else(见图 2-25)
  { while(p->next&&j<i-1) // 寻找第 i 个结点,并令 p 指向其前驱
    { j++; // 计数器+1
      p=p->next; // p 指向下一个结点
    }
    if(!p->next||j>i-1) // 删除位置不合理
      return ERROR; // 删除失败
    q=p->next; // q 指向待删除结点
    p->next=q->next; // 待删结点的前驱指向待删结点的后继
```

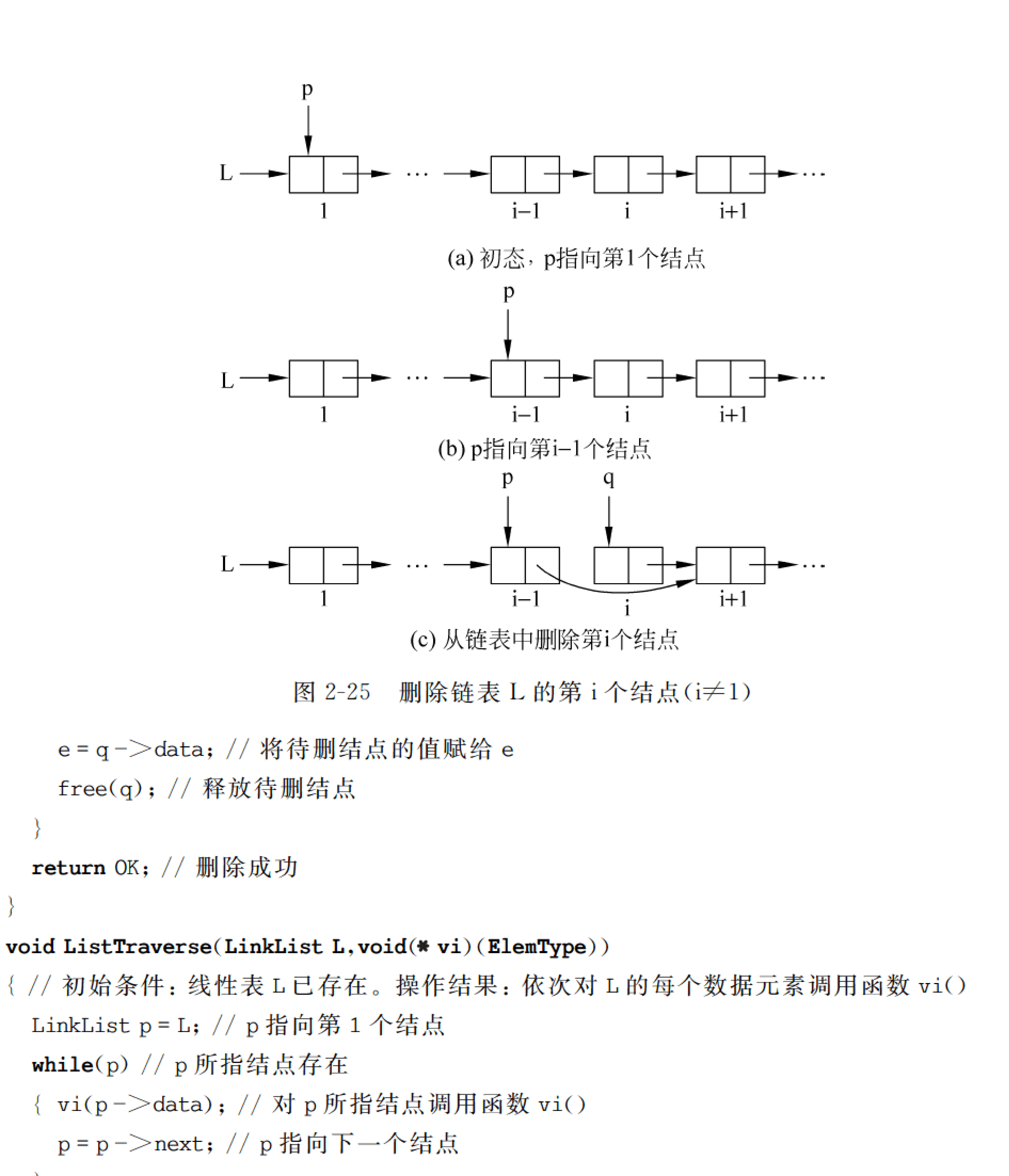



图 2-25 删除链表 L 的第 i 个结点(i≠1)

```
    e = q->data; // 将待删结点的值赋给 e
    free(q); // 释放待删结点
  }
  return OK; // 删除成功
}
void ListTraverse(LinkList L,void(* vi)(ElemType))
{ // 初始条件：线性表 L 已存在。操作结果：依次对 L 的每个数据元素调用函数 vi()
  LinkList p = L; // p 指向第 1 个结点
  while(p) // p 所指结点存在
  { vi(p->data); // 对 p 所指结点调用函数 vi()
    p = p->next; // p 指向下一个结点
  }
  printf("\n");
}
```

```
// bo2-4.cpp 不设头结点的单链表(存储结构由 c2-2.h 定义)的部分基本操作(2 个)
Status PriorElem(LinkList L,ElemType cur_e,ElemType &pre_e)
{ // 初始条件：线性表 L 已存在
  // 操作结果：若 cur_e 是 L 的数据元素，且不是第一个，则用 pre_e 返回它的前驱，返回 OK；
  //           否则操作失败，pre_e 无定义，返回 ERROR
  LinkList q,p = L; // p 指向第 1 个结点
  while(p->next) // p 所指结点有后继
  { q = p->next; // q 指向 p 的后继
    if(q->data == cur_e) // p 的后继为 cur_e
    { pre_e = p->data; // 将 p 所指元素的值赋给 pre_e
      return OK; // 成功返回 OK
    }
    p = q; // p 的后继不为 cur_e，p 向后移
```

```
  }
  return ERROR; // 操作失败,返回 ERROR
}
Status NextElem(LinkList L,ElemType cur_e,ElemType &next_e)
{ // 初始条件:线性表 L 已存在
  // 操作结果:若 cur_e 是 L 的数据元素,且不是最后一个,则用 next_e 返回它的后继,返回 OK,
  //          否则操作失败,next_e 无定义,返回 ERROR
  LinkList p = L; // p 指向第 1 个结点
  while(p->next) // p 所指结点有后继
  { if(p->data == cur_e) // p 所指结点的值为 cur_e
    { next_e = p->next->data; // 将 p 所指结点的后继结点的值赋给 next_e
      return OK; // 成功返回 OK
    }
    p = p->next; // p 指向下一个结点
  }
  return ERROR; // 操作失败,返回 ERROR
}
```

```
// main2-3.cpp 检验 bo2-3.cpp 和 bo2-4.cpp 的主程序
#include"c1.h"
typedef int ElemType; // 定义 ElemType 为整型
#include"c2-2.h" // 线性表的单链表存储结构
#include"bo2-3.cpp" // 不设头结点单链表的基本操作(9 个)
#include"bo2-4.cpp" // 不设头结点单链表的基本操作(2 个)
#include"func2-2.cpp" // 包括 equal()、comp()、print()、print1()和 print2()函数
#include"func2-3.cpp" // 主函数
```

**程序运行结果同 main2-2.cpp 的运行结果。**

和带有头结点的单链表(见图 2-12)相比,不设头结点的单链表(见图 2-20)显得更直观。但不设头结点的单链表在插入和删除第 1 个元素时与插入和删除其他元素时的操作不一样,要改变链表头指针的值。而带有头结点的单链表无论插入和删除第几个元素,其操作都是统一的。这从 bo2-2.cpp 和 bo2-3.cpp 两文件中的 ListInsert()及 ListDelete()函数的区别可看出。不设头结点的单链表在第 7 章中有应用。

顺序存储结构也可以实现链式存储功能。首先,开辟一个充分大的结构体数组。结构体的一个成员存放数据元素,另一个成员(“游标”)存放链表中下一个数据元素在数组中的位置,这称为静态链表。静态链表存于数组中,但链表的输出却不是按数组的顺序输出的,而是由一个指定的位置开始根据游标依次输出的。教科书 10.2.2 节中“表插入排序”就用到了静态链表。c2-3.h 就是这样的一个数据存储结构。algo2-4.cpp 是以 c2-3.h 为数据存储结构,输出教科书中图 2.10 静态链表的示例程序。

```
// c2-3.h 线性表的静态单链表存储结构。在教科书第 31 页(见图 2-26)
#define MAX_SIZE 100 // 链表的最大长度
typedef struct
{ ElemType data;
```

```
  int cur;
}component,SLinkList[MAX_SIZE];
```

component

| data | cur |
|---|---|

SLinkList

| data | cur | |
|---|---|---|
| | | [0] |
| | | [1] |
| | ⋮ | ⋮ |
| | | [MAX_SIZE-2] |
| | | [MAX_SIZE-1] |

图 2-26　静态单链表存储结构

```
// algo2-4.cpp 教科书中图 2.10 静态链表示例
// 第 1 个结点的位置在[0].cur 中。成员 cur 的值为 0,
// 则到链表尾
#include"c1.h"
#define N 6 // 字符串的最大长度+1
typedef char ElemType[N]; // 定义 ElemType 为字符串类型
#include"c2-3.h" // 线性表的静态单链表存储结构
void main()
{
  SLinkList s={{"",1},{"ZHAO",2},{"QIAN",3},{"SUN",4},{"LI",5},{"ZHOU",6},
               {"WU",7},{"ZHENG",8},{"WANG",0}}; // 教科书中图 2.10(a)的状态
  int i=s[0].cur; // i 指示第 1 个结点的位置
  while(i) // 未到链表尾
  { // 输出教科书中图 2.10(a)的状态
    printf("%s",s[i].data); // 输出链表的当前值
    i=s[i].cur; // 找到下一个数据的位置
  }
  printf("\n");
  s[4].cur=9; // 按教科书中图 2.10(b)修改(在"LI"之后插入"SHI")
  s[9].cur=5;
  strcpy(s[9].data,"SHI");
  s[6].cur=8; // 删除"ZHENG"
  i=s[0].cur; // i 指示第 1 个结点的位置
  while(i) // 未到链表尾
  { // 输出教科书中图 2.10(b)的状态
    printf("%s",s[i].data); // 输出链表的当前值
    i=s[i].cur; // 找到下一个数据的位置
  }
  printf("\n");
}
```

**程序运行结果：**

```
ZHAO QIAN SUN LI ZHOU WU ZHENG WANG
ZHAO QIAN SUN LI SHI ZHOU WU WANG
```

如果按照数组的输出方式，教科书中图 2.10(b)的输出应该是：

```
ZHAO QIAN SUN LI ZHOU WU ZHENG WANG SHI
```

algo2-4.cpp 输出不是按数组的顺序输出的，而是像链表一样，由当前结点 cur 成员的值决定下一个结点的位置确定输出，这就是链表的特点。但 algo2-4.cpp 插入新结点完全靠人工判断该结点是否为空闲结点，而不是"自动地"找到空闲结点作为新结点，这不能满足实际应用的需要。为了解决这个问题，要将所有空闲结点链接形成一个备用链表。将数组

下标为0的单元设为备用链表的头结点(这时,链表的头结点就不能是数组下标为0的单元了,一般将数组下标为MAX_SIZE−1的单元,也就是最后一个单元,设为链表的头结点)。这样,在静态数组中实际上有2个链表,一个链表上链接的是线性表的结点,另一个链表(备用链表)上链接的是所有未被使用的结点。静态数组的每一个单元都链接在这2个链表中的一个上。图2-27说明了这种情况。为了图示方便起见,暂定义MAX_SIZE为10。将首尾两个单元分别设为备用链表头结点和链表头结点,如图2-27(a)所示。链表的数据元素依次是1、7、6、2、4。[5]、[3]、[8]三个单元是空闲结点,其中的数据是无效的。cur成员为0的单元是链表的最后一个结点。

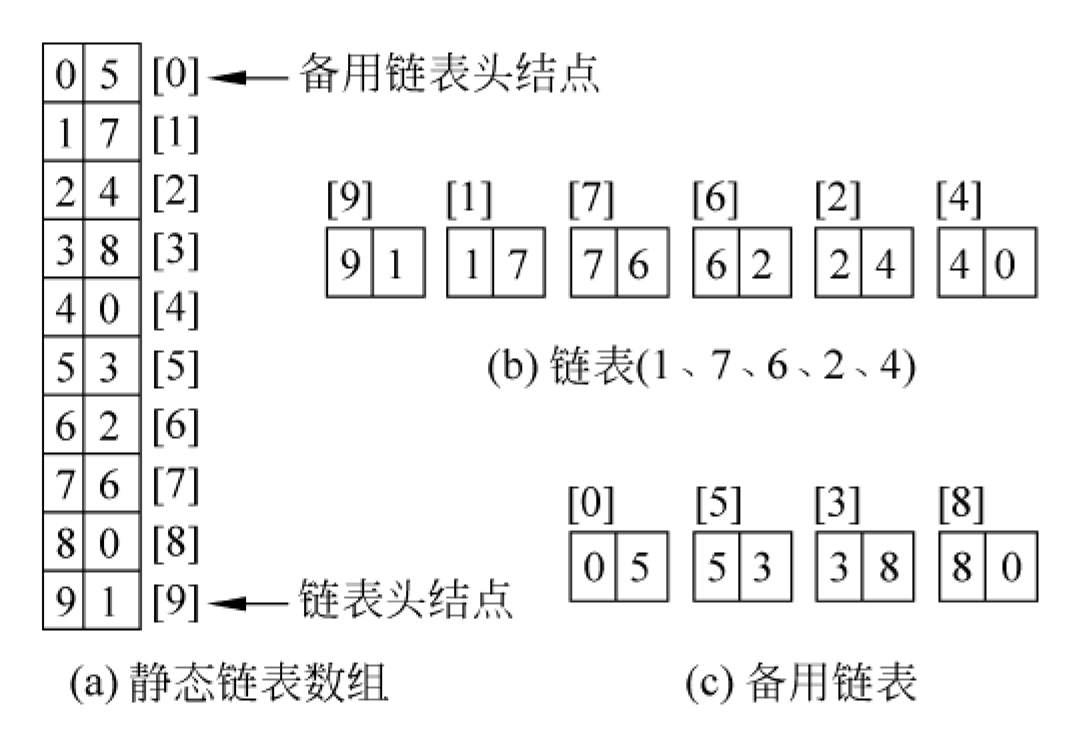



图2-27 静态单链表结构中的2个链表

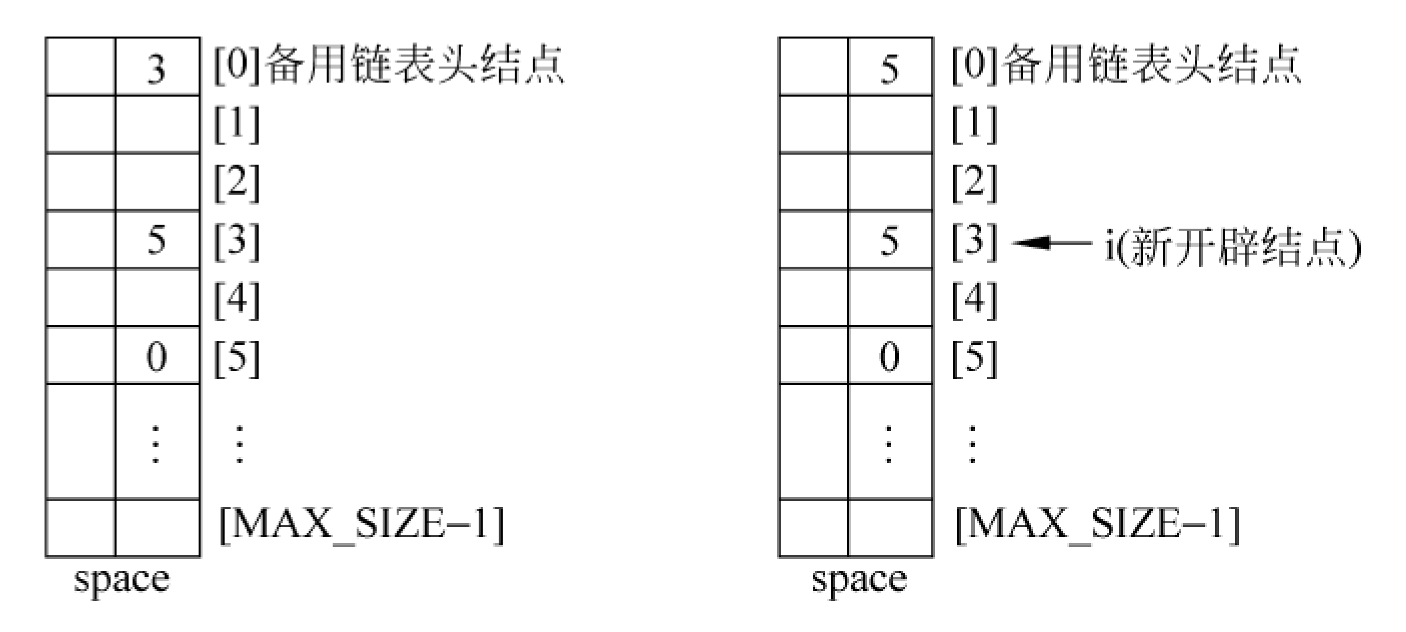
通过上面的初步介绍,可以看出,静态链表和单链表的主要区别有两点:

(1) 单链表的结点通过malloc()函数产生,结点被限制在动态存储区这个大范围内。因此,指示结点位置的指针类型实际上是长整型。而静态链表的结点被限制在静态数组这个小范围内。因此,指示结点位置的指针(cur成员)一般是整型,甚至也可以是字节型,只要MAX_SIZE小于256。从这点上看,静态链表更容易管理。

(2) 单链表中不用的结点通过free()函数释放,释放后的结点由编译软件管理。而静态链表中不用的结点要自己管理(插入到备用链表中)。

当静态链表需要新结点时,把备用链表中的首元结点(由[0].cur指示)从备用链表中删除,作为新结点,插入静态链表。当删除静态链表中的结点时,被删除的结点插入备用链表中,成为备用链表的首元结点。之所以从备用链表删除结点或向备用链表插入结点的操作都在表头进行,是因为这样效率最高。开辟新结点和回收空闲结点的操作如算法2.15和算法2.16所示,在程序bo2-5.cpp中实现。

```
// bo2-5.cpp 静态链表(数据结构由c2-3.h定义)的基本操作(13个)
// 包括算法2.13、算法2.15和算法2.16
#define DestroyList ClearList // DestroyList()和ClearList()的操作是一样的
int Malloc(SLinkList space) // 算法2.15(见图2-28)
{ // 若备用链表非空,则返回分配的结点下标(备用链表的第1个结点),否则返回0
  int i = space[0].cur; // 备用链表第1个结点的位置
  if(i) // 备用链表非空
    space[0].cur = space[i].cur; // 备用链表的头结点指向原备用链表的第2个结点
  return i; // 返回新开辟结点的坐标
}
```

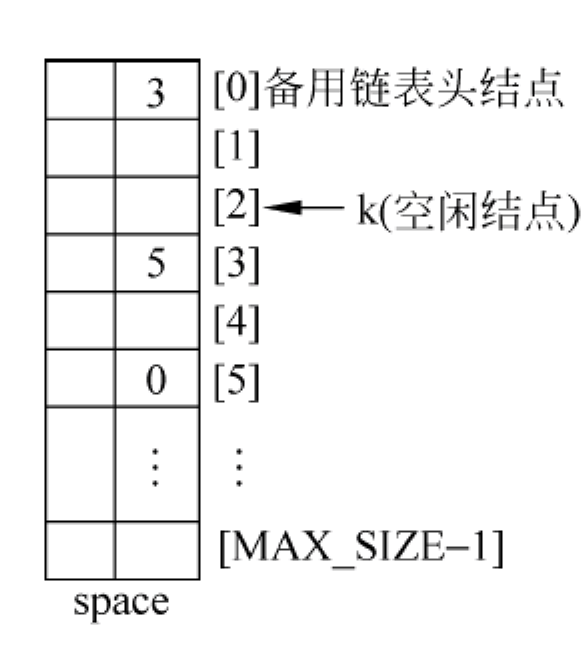



(a) 在调用Malloc()之前，备用链表上共有2个结点：[3]和[5]

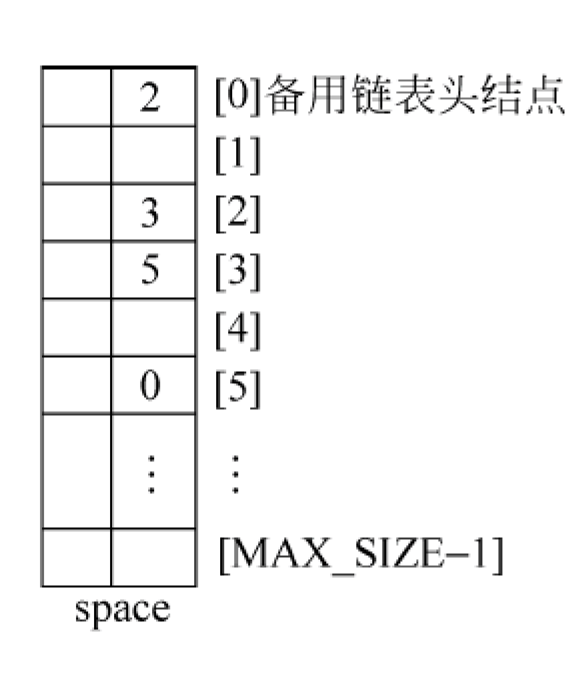
(b) 在调用Malloc()之后，备用链表上只有1个结点[5]

图 2-28　调用 Malloc()示例

```
void Free(SLinkList space,int k) // 算法 2.16(见图 2-29)
{ // 将下标为 k 的空闲结点回收到备用链表中(成为备用链表的第 1 个结点)
  space[k].cur = space[0].cur; // 回收结点的“游标”指向备用链表的第 1 个结点
  space[0].cur = k; // 备用链表的头结点指向新回收的结点
}
void InitList(SLinkList L)
{ // 构造一个空的链表 L,表头为 L 的最后一个单元 L[MAX_SIZE - 1],其余单元链成
  // 一个备用链表,表头为 L 的第一个单元 L[0],“0”表示空指针(见图 2-30)
```




(a) 在调用Free()之前，备用链表上有2个结点：[3]和[5]




(b) 在调用Free()之后，备用链表上有3个结点[2]、[3]和[5]

图 2-29　调用 Free()示例

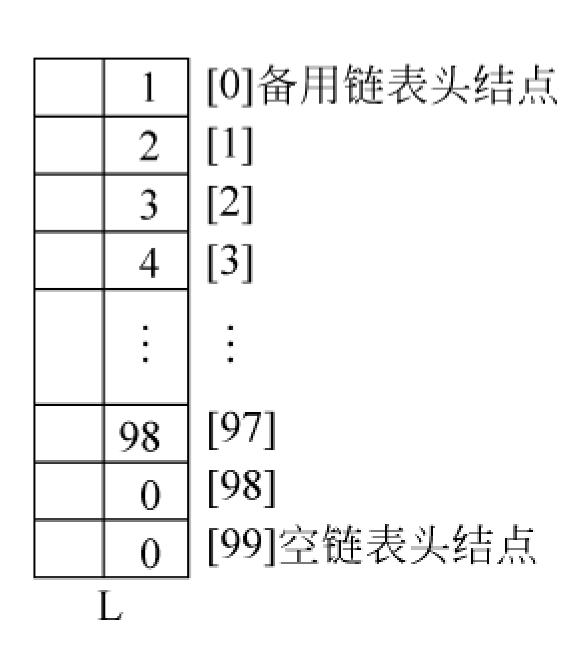



图 2-30　构造空链表 L

```
  int i;
  L[MAX_SIZE - 1].cur = 0; // L 的最后一个单元为空链表的表头
  for(i = 0;i<MAX_SIZE - 2;i ++ ) // 将其余单元链接成以 L[0]为表头的备用链表
    L[i].cur = i + 1;
  L[MAX_SIZE - 2].cur = 0;
}
void ClearList(SLinkList L)
{ // 初始条件:线性表 L 已存在。操作结果:将 L 重置为空表
  int j,k,i = L[MAX_SIZE - 1].cur; // i 指示链表第 1 个结点的位序
  L[MAX_SIZE - 1].cur = 0; // 链表空
  k = L[0].cur; // 备用链表第 1 个结点的位置
  L[0].cur = i; // 把链表的结点连到备用链表的表头
```

```
    while(i) // 未到链表尾
    { j = i; // j 指示当前结点的位序
      i = L[i].cur; // i 指向下一个结点的位序
    }
    L[j].cur = k; // 备用链表的第 1 个结点接到链表的尾部
  }
  Status ListEmpty(SLinkList L)
  { // 若 L 是空表,返回 TRUE,否则返回 FALSE
    if(L[MAX_SIZE - 1].cur == 0) // 空表(表头结点的 cur 域为 0)
      return TRUE;
    else
      return FALSE;
  }
  int ListLength(SLinkList L)
  { // 返回 L 中数据元素个数
    int j = 0,i = L[MAX_SIZE - 1].cur; // i 指示链表的第 1 个结点的位序
    while(i) // 未到静态链表尾
    { i = L[i].cur; // i 指向下一个结点
      j ++; // 计数器 + 1
    }
    return j;
  }
  Status GetElem(SLinkList L,int i,ElemType &e)
  { // 用 e 返回 L 中第 i 个元素的值
    int m,k = MAX_SIZE - 1; // k 指示表头结点的位序
    if(i<1||i>ListLength(L)) // 不存在第 i 个元素
      return ERROR;
    for(m = 1;m<= i;m ++ ) // k 向后移动到第 i 个元素处
      k = L[k].cur; // 指向下一个元素
    e = L[k].data; // 将第 i 个元素的值赋给 e
    return OK;
  }
  int LocateElem(SLinkList L,ElemType e) // 算法 2.13(有改动)
  { // 在静态单链线性表 L 中查找第 1 个值为 e 的元素。若找到,则返回它在 L 中的位序,否则返回 0
    // (与其他 LocateElem()的定义不同)
    int i = L[MAX_SIZE - 1].cur; // i 指示表中第 1 个结点的位序
    while(i&&L[i].data! = e) // 在表中顺链查找(e 不能是字符串),找到或已到表尾,结束循环
      i = L[i].cur; // 指向下一个元素
    return i;
  }
  Status PriorElem(SLinkList L,ElemType cur_e,ElemType &pre_e)
  { // 初始条件:线性表 L 已存在
    // 操作结果:若 cur_e 是 L 的数据元素,且不是第一个,则用 pre_e 返回它的前驱;
    //           否则操作失败,pre_e 无定义
    int j,i = L[MAX_SIZE - 1].cur; // i 指示链表第 1 个结点的位置
```

```
  do // 向后移动结点
  { j = i; // j 指向 i 所指元素
    i = L[i].cur; // i 指向下一个元素
  }while(i&&cur_e! = L[i].data); // i 所指元素存在且其值不是 cur_e,继续循环
  if(i) // 找到该元素
  { pre_e = L[j].data; // j 所指元素是 i 所指元素的前驱元素,赋给 pre_e
    return OK;
  }
  return ERROR; // 未找到该元素,或其无前驱
}
Status NextElem(SLinkList L,ElemType cur_e,ElemType &next_e)
{ // 初始条件:线性表 L 已存在
  // 操作结果:若 cur_e 是 L 的数据元素,且不是最后一个,则用 next_e 返回它的后继;
  //           否则操作失败,next_e 无定义
  int j,i = LocateElem(L,cur_e); // 在 L 中查找第 1 个值为 cur_e 的元素的位置
  if(i) // L 中存在元素 cur_e
  { j = L[i].cur; // j 指示 cur_e 的后继的位置
    if(j) // cur_e 有后继
    { next_e = L[j].data; // 将 cur_e 的后继赋给 next_e
      return OK; // cur_e 元素有后继
    }
  }
  return ERROR; // L 不存在 cur_e 元素,或 cur_e 元素无后继
}
Status ListInsert(SLinkList L,int i,ElemType e)
{ // 在 L 中第 i 个元素之前插入新的数据元素 e(见图 2-31)
```

| data | cur | |
|---|---|---|
| | 3 | [0]备用链表头结点 |
| 6 | 2 | [1] |
| 25 | 0 | [2] |
| | 4 | [3] |
| | ⋮ | ⋮ |
| | 98 | [97] |
| | 0 | [98] |
| | 1 | [99]链表头结点 |

L

参数:
i=2
e=33

(a) 在调用ListInsert()之前,链表L上有2个元素:6和25

| data | cur | |
|---|---|---|
| | 4 | [0]备用链表头结点 |
| 6 | 3 | [1] |
| 25 | 0 | [2] |
| 33 | 2 | [3] |
| | ⋮ | ⋮ |
| | 98 | [97] |
| | 0 | [98] |
| | 1 | [99]链表头结点 |

L

(b) 在调用ListInsert()之后,链表L上有3个元素:6、33和25

图 2-31 调用 ListInsert()示例

```
  int m,j,k = MAX_SIZE - 1; // k 指示表头结点的位序
  if(i<1||i>ListLength(L) + 1) // i 值不合法
    return ERROR;
  j = Malloc(L); // 申请新单元
  if(j) // 申请成功
  { L[j].data = e; // 将 e 赋值给新单元
    for(m = 1;m<i;m ++ ) // k 向后移动 i - 1 个结点,使 k 指示第 i - 1 个结点
```

```
        k = L[k].cur; // 指向下一个结点
      L[j].cur = L[k].cur; // 新单元指向第 i - 1 个元素后面的元素(第 i 个元素)
      L[k].cur = j; // 第 i - 1 个元素指向新单元
      return OK;
    }
    return ERROR;
}
Status ListDelete(SLinkList L,int i,ElemType &e)
{ // 删除在 L 中第 i 个数据元素 e,并返回其值(见图 2-32)
```

| | | |
|---|---|---|
| | 3 | [0]备用链表头结点 |
| 6 | 2 | [1] |
| 25 | 0 | [2] |
| | 4 | [3] |
| | ⋮ | ⋮ |
| | 98 | [97] |
| | 0 | [98] |
| | 1 | [99]链表头结点 |

L

参数：
i=2

(a) 在调用ListDelete()之前，链表L上有2个元素：6和25

| | | |
|---|---|---|
| | 2 | [0]备用链表头结点 |
| 6 | 0 | [1] |
| 25 | 3 | [2] |
| | 4 | [3] |
| | ⋮ | ⋮ |
| | 98 | [97] |
| | 0 | [98] |
| | 1 | [99]链表头结点 |

L

返回值：
e=25

(b) 在调用ListDelete()之后，链表L上有1个元素：6

图 2-32　调用 ListDelete()示例

```
    int j,k = MAX_SIZE - 1; // k 指示表头结点的位序
    if(i<1||i>ListLength(L)) // i 值不合法
      return ERROR;
    for(j = 1;j<i;j ++ ) // 移动 i - 1 个元素,使 k 指向第 i - 1 个元素
      k = L[k].cur; // 指向下一个元素
    j = L[k].cur; // 待删除元素(第 i 个元素)的位置赋给 j
    L[k].cur = L[j].cur; // 使第 i - 1 个元素指向待删除元素的后继元素
    e = L[j].data; // 待删除元素的值赋给 e
    Free(L,j); // 释放待删除结点(回收到备用链表中)
    return OK;
}
void ListTraverse(SLinkList L,void(*visit)(ElemType))
{ // 初始条件:线性表 L 已存在。操作结果:依次对 L 的每个数据元素调用函数 visit()
    int i = L[MAX_SIZE - 1].cur; // i 指示第 1 个元素的位序
    while(i) // 未到静态链表尾
    { visit(L[i].data); // 对当前元素调用 visit()
      i = L[i].cur; // 指向下一个元素
    }
    printf("\n");
}
```

```
// main2-4.cpp 检验 bo2-5.cpp 的主程序
#include"c1.h"
typedef int ElemType; // 定义 ElemType 为整型
```

```
#include"c2-3.h" // 静态单链表的存储结构
#include"bo2-5.cpp" // 静态链表的基本操作(13 个),包括算法 2.13、算法 2.15 和算法 2.16
#include"func2-2.cpp" // 包括 equal()、comp()、print()、print1()和 print2()函数
typedef SLinkList LinkList; // 定义 func2-3.cpp 中的 LinkList 类型为 SLinkList 类型
#define SLL // 定义了 SLL(静态链表标志),使 func2-3.cpp 选择执行静态链表特有的函数
#include"func2-3.cpp" // 主函数
```

**程序运行结果：**

```
在 L 的表头依次插入 1～5 后,L = 5 4 3 2 1
L 是否空? i = 0(1:是 0:否),表 L 的长度 = 5
清空 L 后,L =
L 是否空? i = 1(1:是 0:否),表 L 的长度 = 0
在 L 的表尾依次插入 1～10 后,L = 1 2 3 4 5 6 7 8 9 10
没有值为 0 的元素,值为 1 的元素的位序为 5
元素 1 无前驱,元素 2 的前驱为 1
元素 9 的后继为 10,元素 10 无后继
删除第 11 个元素失败(不存在此元素)。删除第 10 个元素成功,其值为 10
依次输出 L 的元素:1 2 3 4 5 6 7 8 9
```

## 2.3.2　循环链表

单链的循环链表结点的存储结构和单链表结点的存储结构一样。所不同的是：最后一个结点的 next 域指向头结点,而不是“空”。这样,由表尾很容易找到表头。但若链表较长,则由表头找到表尾较费时。因而,单循环链表往往设立尾指针而不是头指针,如图 2-33 所示。这在两个链表首尾相连合并成一个链表时非常方便。bo2-6.cpp 是设立尾指针的单循环链表的基本操作。

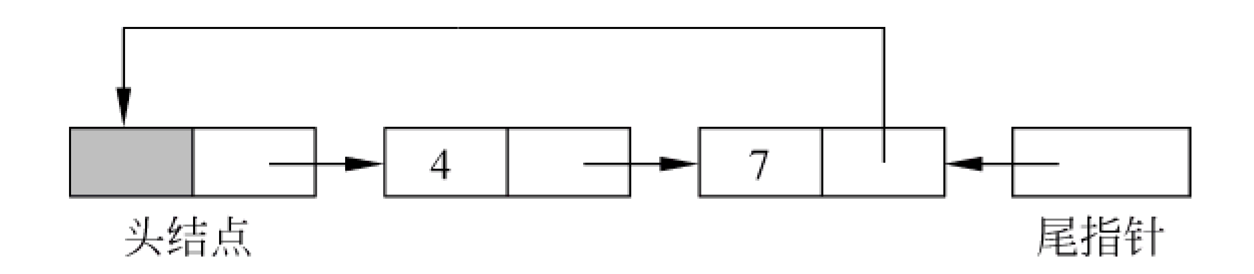



图 2-33　设立尾指针且具有 2 个结点(4,7)的单循环链表

```
// bo2-6.cpp 设立尾指针的单循环链表(存储结构由 c2-2.h 定义)的 12 个基本操作
void InitList(LinkList &L)
{ // 操作结果:构造一个空的线性表 L(见图 2-34)
  L = (LinkList)malloc(sizeof(LNode)); // 产生头结点,并使 L 指向此头结点
  if(!L) // 存储分配失败
    exit(OVERFLOW);
  L->next = L; // 头结点的指针域指向头结点
}
void ClearList(LinkList &L) // 改变 L
{ // 初始条件:线性表 L 已存在。操作结果:将 L 重置为空表(见图 2-34)
  LinkList p,q;
```

头结点　L

图 2-34　空(仅有头结点)的单循环链表 L

```
    L = L->next; // L指向头结点
    p = L->next; // p指向第1个结点
    while(p! = L) // 未到表尾
    { q = p->next; // q指向p的后继结点
      free(p); // 释放p所指结点
      p = q; // p指向q所指结点
    }
    L->next = L; // 头结点指针域指向自身(头结点)
}
void DestroyList(LinkList &L)
{ // 操作结果:销毁线性表L(见图2-35)
    ClearList(L); // 将表L重置为空表
    free(L); // 释放L所指结点(头结点)
    L = NULL; // L不指向任何存储单元
}
```

| NULL |
|---|
| L |

图 2-35 线性表 L 被销毁

```
Status ListEmpty(LinkList L)
{ // 初始条件:线性表L已存在。操作结果:若L为空表,则返回TRUE,否则返回FALSE
    if(L->next == L) // 空
      return TRUE;
    else
      return FALSE;
}
int ListLength(LinkList L)
{ // 初始条件:L已存在。操作结果:返回L中数据元素个数
    int i = 0;
    LinkList p = L->next; // p指向头结点
    while(p! = L) // 未到表尾
    { i++; // 计数器+1
      p = p->next; // p指向下一个结点
    }
    return i;
}
Status GetElem(LinkList L,int i,ElemType &e)
{ // 当第i个元素存在时,其值赋给e并返回OK,否则返回ERROR
    int j = 1; // 初始化,j为计数器,初值为1
    LinkList p = L->next->next; // p指向第1个结点
    if(i<= 0||i>ListLength(L)) // 第i个元素不存在
      return ERROR;
    while(j<i) // 顺指针向后查找,直到p指向第i个元素
    { j++; // 计数器+1
      p = p->next; // p指向下一个结点
    }
    e = p->data; // 第i个元素赋给e
    return OK;
```

```
}
int LocateElem(LinkList L,ElemType e,Status(* compare)(ElemType,ElemType))
{ // 初始条件：线性表 L 已存在，compare()是数据元素判定函数
  // 操作结果：返回 L 中第 1 个与 e 满足关系 compare()的数据元素的位序。
  //           若这样的数据元素不存在，则返回值为 0
  int i = 0; // 计数器初值为 0
  LinkList p = L->next->next; // p 指向第 1 个结点
  while(p! = L->next) // p 未指向头结点
  { i++; // 计数器 +1
    if(compare(p->data,e)) // 找到这样的数据元素
      return i; // 返回其位序
    p = p->next; // p 指向下一个结点
  }
  return 0; // 满足关系的数据元素不存在
}
Status PriorElem(LinkList L,ElemType cur_e,ElemType &pre_e)
{ // 初始条件：线性表 L 已存在
  // 操作结果：若 cur_e 是 L 的数据元素，且不是第一个，则用 pre_e 返回它的前驱，返回 OK，
  //           否则操作失败，pre_e 无定义，返回 ERROR
  LinkList q,p = L->next->next; // p 指向第 1 个结点
  q = p->next; // q 指向 p 的后继
  while(q! = L->next) // p 未到表尾(q 未指向头结点)
  { if(q->data == cur_e) // p 的后继为 cur_e
    { pre_e = p->data; // 将 p 所指元素的值赋给 pre_e
      return OK; // 成功返回 OK
    }
    p = q; // p 的后继不为 cur_e,p 向后移
    q = q->next; // q 指向 p 的后继
  }
  return ERROR; // 操作失败，返回 ERROR
}
Status NextElem(LinkList L,ElemType cur_e,ElemType &next_e)
{ // 初始条件：线性表 L 已存在
  // 操作结果：若 cur_e 是 L 的数据元素，且不是最后一个，则用 next_e 返回它的后继，返回 OK,
  //           否则操作失败，next_e 无定义，返回 ERROR
  LinkList p = L->next->next; // p 指向第 1 个结点
  while(p! = L) // p 未到表尾
  { if(p->data == cur_e) // p 所指结点的值为 cur_e
    { next_e = p->next->data; // 将 p 所指结点的后继结点的值赋给 next_e
      return OK; // 成功返回 OK
    }
    p = p->next; // p 指向下一个结点
  }
  return ERROR; // 操作失败，返回 ERROR
}
```

```
Status ListInsert(LinkList &L,int i,ElemType e)
{ // 在 L 的第 i 个位置之前插入元素 e(改变 L)
  LinkList p = L->next,s; // p 指向头结点
  int j = 0; // 计数器初值为 0
  if(i<= 0||i>ListLength(L) + 1) // i 值不合法
    return ERROR; // 插入失败
  while(j<i - 1) // 寻找第 i - 1 个结点
  { j++; // 计数器 + 1
    p = p->next; // p 指向下一个结点
  }
  s = (LinkList)malloc(sizeof(LNode)); // 生成新结点,以下将其插入 L 中
  s->data = e; // 将 e 赋给新结点
  s->next = p->next; // 新结点指向原第 i 个结点
  p->next = s; // 原第 i - 1 个结点指向新结点
  if(p == L) // 插在表尾(见图 2-36)
    L = s; // L 指向新的尾结点
  return OK; // 插入成功
}
```



(a) 初态,p指向头结点

(b) p指向第i–1个结点(尾结点)

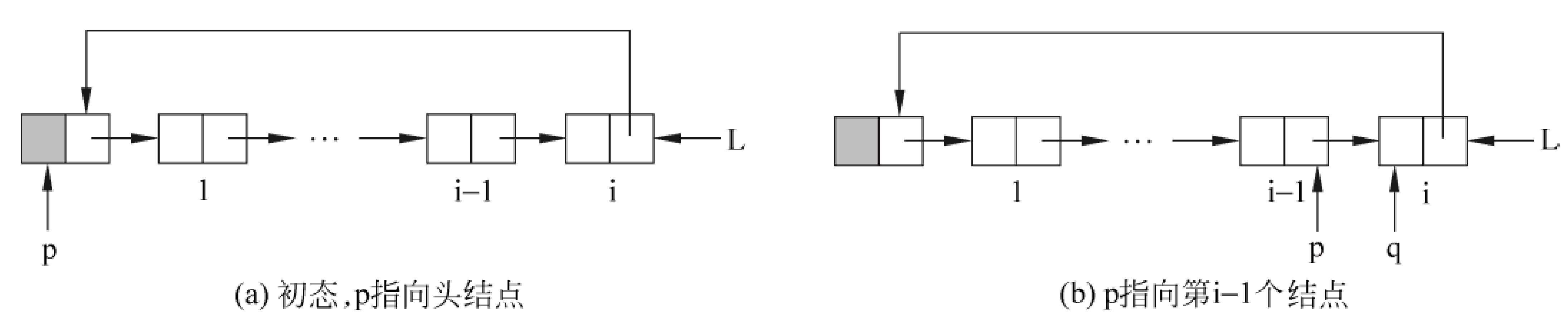


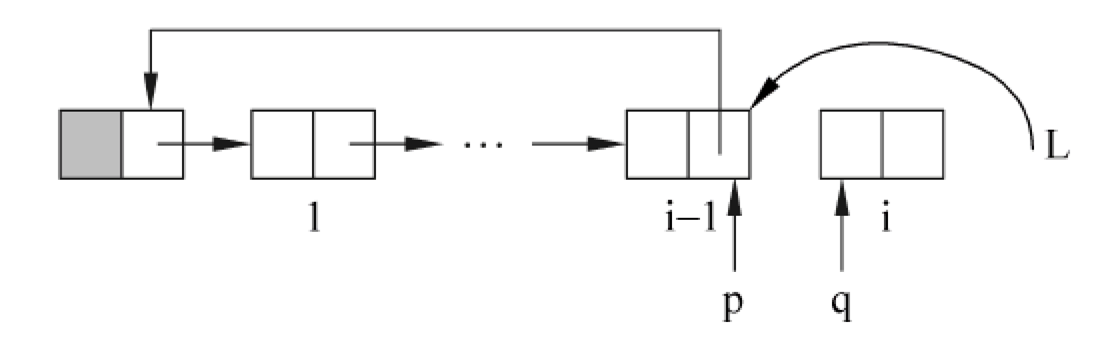
(c) 插入s为尾结点

图 2-36 在链表 L 的表尾插入元素 e

```
Status ListDelete(LinkList &L,int i,ElemType &e)
{ // 删除 L 的第 i 个元素,并由 e 返回其值(改变 L)
  LinkList q,p = L->next; // p 指向头结点
  int j = 0; // 计数器初值为 0
  if(i<= 0||i>ListLength(L)) // 第 i 个元素不存在
    return ERROR; // 删除失败
  while(j<i - 1) // 寻找第 i - 1 个结点
  { j++; // 计数器 + 1
    p = p->next; // p 指向下一个结点
  }
  q = p->next; // q 指向待删除结点,p 的后继
  p->next = q->next; // 待删结点的前驱指向待删结点的后继
```

```
    e = q->data; // 将待删结点的值赋给 e
    if(L == q) // 删除的是表尾元素(见图 2-37)
      L = p; // L 指向新的表尾元素
```




(c) 从链表上删除第i个结点

图 2-37　删除链表 L 的表尾结点

```
    free(q); // 释放待删除结点
    return OK; // 删除成功
  }
  void ListTraverse(LinkList L,void(*vi)(ElemType))
  { // 初始条件:L 已存在。操作结果:依次对 L 的每个数据元素调用函数 vi()
    LinkList p = L->next->next; // p 指向首元结点
    while(p! = L->next) // p 不指向头结点
    { vi(p->data); // 对 p 所指结点调用函数 vi()
      p = p->next; // p 指向下一个结点
    }
    printf("\n");
  }
```

```
  // main2-5.cpp 单循环链表,检验 bo2-6.cpp 的主程序
  #include"c1.h"
  typedef int ElemType; // 定义 ElemType 为整型
  #include"c2-2.h" // 线性表的单链表存储结构
  #include"bo2-6.cpp" // 设立尾指针的单循环链表存储结构的 12 个基本操作
  #include"func2-2.cpp" // 包括 equal()、comp()、print()、print1()和 print2()函数
  #include"func2-3.cpp" // 主函数
```

程序运行结果同 main2-2.cpp 的运行结果。

## 2.3.3　双向链表

```
// c2-4.h 线性表的双向链表存储结构。在教科书第 35 页(见图 2-38)
typedef struct DuLNode
{ ElemType data;
```

```
  DuLNode *prior, *next;
}DuLNode, *DuLinkList;
```

双向链表(见图 2-39)的每个结点有 2 个指针,一个指向结点的前驱,另一个指向结点的后继。所以,在双向链表中,不仅可以容易地找到后继结点,也很容易找到前驱结点。

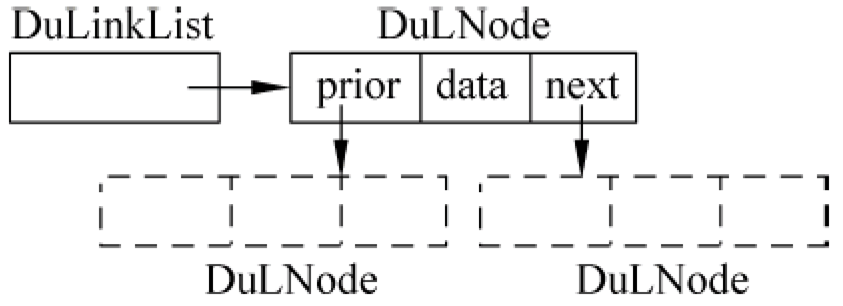


图 2-38　线性表的双向链表存储结构

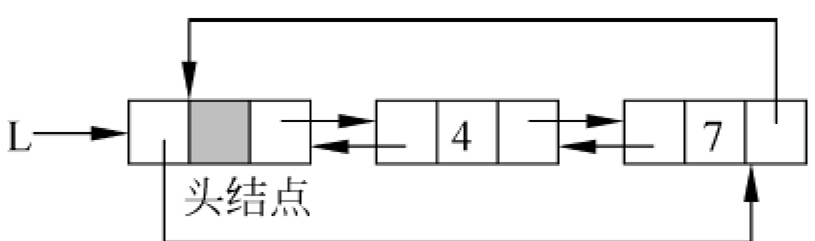


图 2-39　带有头结点且具有 2 个结点(4,7)的双向循环链表 L

```
// bo2-7.cpp 带头结点的双向循环链表(存储结构由 c2-4.h 定义)的基本操作(14 个)
// 包括算法 2.18 和算法 2.19
void InitList(DuLinkList &L)
{ // 产生空的双向循环链表 L(见图 2-40)
  L = (DuLinkList)malloc(sizeof(DuLNode));
  if(L)
    L->next = L->prior = L;
  else
    exit(OVERFLOW);
}
```

L　头结点

图 2-40　空的双向循环链表 L

```
void ClearList(DuLinkList L) // 不改变 L
{ // 初始条件:L 已存在。操作结果:将 L 重置为空表(见图 2-40)
  DuLinkList p = L->next; // p 指向第 1 个结点
  while(p! = L) // p 未指向头结点
  { p = p->next; // p 指向下一个结点
    free(p->prior); // 释放 p 的前驱结点
  }
  L->next = L->prior = L; // 头结点的两个指针域均指向自身
}
void DestroyList(DuLinkList &L)
{ // 操作结果:销毁双向循环链表 L(见图 2-41)
  ClearList(L); // 将 L 重置为空表(见图 2-40)
  free(L); // 释放头结点
  L = NULL; // L 不指向任何存储单元
}
```

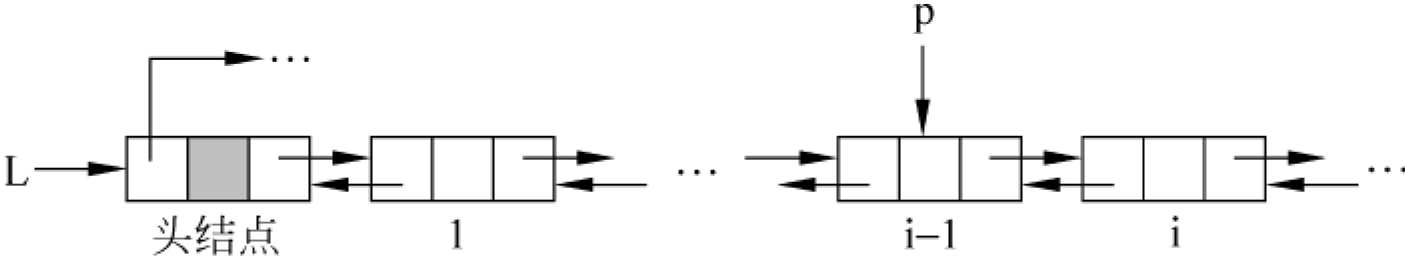
NULL
L

图 2-41　销毁双向循环链表 L

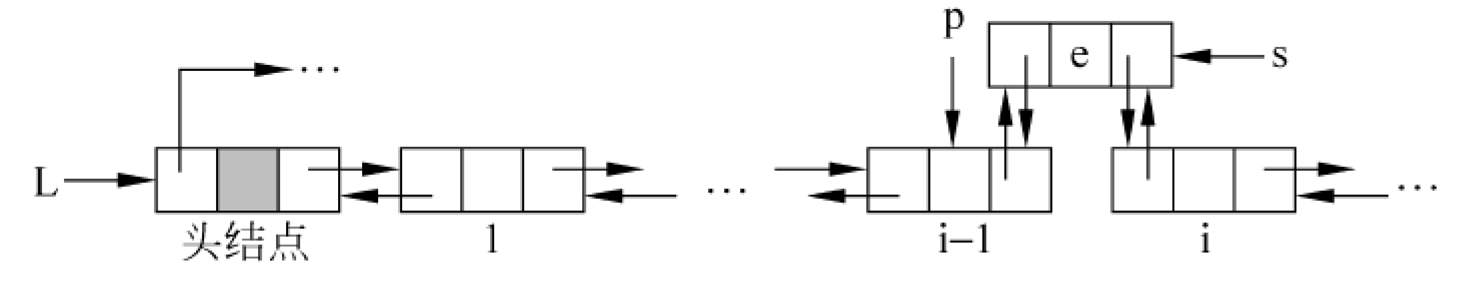
```
Status ListEmpty(DuLinkList L)
{ // 初始条件:线性表 L 已存在。操作结果:若 L 为空表,则返回 TRUE,否则返回 FALSE
  if(L->next == L&&L->prior == L)
    return TRUE;
  else
    return FALSE;
}
int ListLength(DuLinkList L)
```

```
{ // 初始条件：L已存在。操作结果：返回L中数据元素个数
  int i=0; // 计数器初值为0
  DuLinkList p=L->next; // p指向第1个结点
  while(p!=L) // p未指向头结点
  { i++; // 计数器+1
    p=p->next; // p指向下一个结点
  }
  return i;
}
Status GetElem(DuLinkList L,int i,ElemType &e)
{ // 当第i个元素存在时,其值赋给e并返回OK;否则返回ERROR
  int j=1; // 计数器初值为1
  DuLinkList p=L->next; // p指向第1个结点
  while(p!=L&&j<i) // 顺指针向后查找,直到p指向第i个元素或p指向头结点
  { j++; // 计数器+1
    p=p->next; // p指向下一个结点
  }
  if(p==L||j>i) // 第i个元素不存在
    return ERROR; // 查找失败
  e=p->data; // 取第i个元素赋给e
  return OK; // 查找成功
}
int LocateElem(DuLinkList L,ElemType e,Status(*compare)(ElemType,ElemType))
{ // 初始条件：L已存在,compare()是数据元素判定函数
  // 操作结果：返回L中第1个与e满足关系compare()的数据元素的位序。
  //           若这样的数据元素不存在,则返回值为0
  int i=0; // 计数器初值为0
  DuLinkList p=L->next; // p指向第1个元素
  while(p!=L) // p未指向头结点
  { i++; // 计数器+1
    if(compare(p->data,e)) // 找到这样的数据元素
      return i; // 返回其位序
    p=p->next; // p指向下一个结点
  }
  return 0; // 满足关系的数据元素不存在
}
Status PriorElem(DuLinkList L,ElemType cur_e,ElemType &pre_e)
{ // 操作结果：若cur_e是L的数据元素,且不是第一个,则用pre_e返回它的前驱,返回OK;
  //           否则操作失败,pre_e无定义,返回ERROR
  DuLinkList p=L->next->next; // p指向第2个元素
  while(p!=L) // p未指向头结点
  { if(p->data==cur_e) // p指向值为cur_e的结点
    { pre_e=p->prior->data; // 将p的前驱结点的值赋给pre_e
      return OK; // 成功返回OK
    }
```

```
      p = p->next; // p 指向下一个结点
    }
    return ERROR; // 操作失败,返回 ERROR
}
Status NextElem(DuLinkList L,ElemType cur_e,ElemType &next_e)
{ // 操作结果:若 cur_e 是 L 的数据元素,且不是最后一个,则用 next_e 返回它的后继,返回 OK,
  //           否则操作失败,next_e 无定义,返回 ERROR
  DuLinkList p = L->next->next; // p 指向第 2 个元素
  while(p! = L) // p 未指向头结点
  { if(p->prior->data == cur_e) // p 所指结点的前驱的值为 cur_e
    { next_e = p->data; // 将 p 所指结点(cur_e 的后继)的值赋给 next_e
      return OK; // 成功返回 OK
    }
    p = p->next; // p 指向下一个结点
  }
  return ERROR; // 操作失败,返回 ERROR
}
DuLinkList GetElemP(DuLinkList L,int i) // 新增
{ // 在双向链表 L 中返回第 i 个元素的地址。i 为 0,返回头结点的地址。若第 i 个元素不存在,
  // 返回 NULL(算法 2.18 和算法 2.19 要调用的函数)
  int j;
  DuLinkList p = L; // p 指向头结点
  if(i<0||i>ListLength(L)) // i 值不合法
    return NULL;
  for(j = 1;j<= i;j ++ ) // p 指向第 i 个结点
    p = p->next; // p 指向下一个结点
  return p;
}
Status ListInsert(DuLinkList L,int i,ElemType e)
{ // 在带头结点的双链循环线性表 L 中第 i 个位置之前插入元素 e,i 的合法值为 1≤i≤表长 + 1
  // 改进算法 2.18,否则无法在第表长 + 1 个结点之前插入元素(见图 2-42)
```




(a) 初态,p指向第i-1个结点




(b) 在第i个结点之前插入值为e的结点

图 2-42 在链表 L 的第 i 个位置之前插入元素 e

```
  DuLinkList p,s;
  if(i<1||i>ListLength(L) + 1) // i 值不合法
```

```
    return ERROR; // 失败返回 ERROR
  p = GetElemP(L,i - 1); // 在 L 中确定第 i 个结点前驱的位置指针 p
  if(!p) // p = NULL,即第 i 个结点的前驱不存在(设头结点为第 1 个结点的前驱)
    return ERROR; // 失败返回 ERROR
  s = (DuLinkList)malloc(sizeof(DuLNode)); // 生成新结点
  if(!s)
    return ERROR; // 生成新结点失败返回 ERROR
  s->data = e; // 将 e 赋给新结点
  s->prior = p; // 新结点的前驱为第 i - 1 个结点
  s->next = p->next; // 新结点的后继为第 i 个结点
  p->next->prior = s; // 第 i 个结点的前驱指向新结点
  p->next = s; // 第 i - 1 个结点的后继指向新结点
  return OK; // 成功返回 OK
}
Status ListDelete(DuLinkList L,int i,ElemType &e) // 算法 2.19
{ // 删除带头结点的双链循环线性表 L 的第 i 个元素,i 的合法值为 1≤i≤表长(见图 2-43)
```



(a) 初态,p指向第i个结点



(b) 从链表中删除第i个结点

图 2-43 删除链表 L 的第 i 个结点

```
  DuLinkList p;
  if(i<1) // i 值不合法
    return ERROR; // 删除失败
  p = GetElemP(L,i); // 在 L 中确定第 i 个元素的位置指针 p
  if(!p) // p = NULL,即第 i 个元素不存在
    return ERROR; // 删除失败
  e = p->data; // 将第 i 个元素的值赋给 e
  p->prior->next = p->next; // 第 i - 1 个结点的后继指向原第 i + 1 个结点
  p->next->prior = p->prior; // 原第 i + 1 个结点的前驱指向第 i - 1 个结点
  free(p); // 释放第 i 个结点
  return OK; // 删除成功
}
void ListTraverse(DuLinkList L,void(* visit)(ElemType))
{ // 由双链循环线性表 L 的头结点出发,正序对每个数据元素调用函数 visit()
  DuLinkList p = L->next; // p 指向首元结点
  while(p! = L) // p 未指向头结点
  { visit(p->data); // 对 p 所指结点调用函数 visit()
```

```
    p = p->next; // p 指向下一个结点
  }
  printf("\n");
}
void ListTraverseBack(DuLinkList L,void( * visit)(ElemType))
{ // 由双链循环线性表 L 的头结点出发,逆序对每个数据元素调用函数 visit()。新增
  DuLinkList p = L->prior; // p 指向尾结点
  while(p! = L) // p 未指向头结点
  { visit(p->data); // 对 p 所指结点调用函数 visit()
    p = p->prior; // p 指向前一个结点
  }
  printf("\n");
}
```

```
// main2-6.cpp 检验 bo2-7.cpp 的主程序
#include"c1.h"
typedef int ElemType; // 定义 ElemType 为整型
#include"c2-4.h" // 线性表的双向链表存储结构
#include"bo2-7.cpp" // 带头结点的双向循环链表存储结构的基本操作(14 个)
#include"func2-2.cpp" // 包括 equal()、comp()、print()、print1()和 print2()函数
void main()
{
  DuLinkList L;
  int i,n = 4;
  Status j;
  ElemType e;
  InitList(L); // 初始化线性表 L
  for(i = 1;i<= 5;i ++ ) // 依次插入 1～5
    ListInsert(L,i,i); // 在第 i 个结点之前插入 i
  printf("逆序输出链表:");
  ListTraverseBack(L,print); // 逆序输出
  j = GetElem(L,2,e); // 将链表的第 2 个元素赋值给 e
  if(j)
    printf("链表的第 2 个元素值为 %d\n",e);
  else
    printf("不存在第 2 个元素\n");
  i = LocateElem(L,n,equal);
  if(i)
    printf("等于 %d 的元素是第 %d 个\n",n,i);
  else
    printf("没有等于 %d 的元素\n",n);
  j = PriorElem(L,n,e);
  if(j)
    printf("%d 的前驱是 %d,",n,e);
  else
```

```
    printf("不存在 %d 的前驱\n",n);
  j = NextElem(L,n,e);
  if(j)
    printf("%d 的后继是 %d\n",n,e);
  else
    printf("不存在 %d 的后继\n",n);
  ListDelete(L,2,e); // 删除并释放第 2 个结点
  printf("删除第 2 个结点,值为 %d,其余结点为 ",e);
  ListTraverse(L,print); // 正序输出
  printf("链表的元素个数为 %d,",ListLength(L));
  printf("链表是否空? %d(1:是 0:否)\n",ListEmpty(L));
  ClearList(L); // 清空链表
  printf("清空后,链表是否空? %d(1:是 0:否)\n",ListEmpty(L));
  DestroyList(L);
}
```

**程序运行结果：**

```
逆序输出链表：5 4 3 2 1
链表的第 2 个元素值为 2
等于 4 的元素是第 4 个
4 的前驱是 3,4 的后继是 5
删除第 2 个结点,值为 2,其余结点为 1 3 4 5
链表的元素个数为 4,链表是否空? 0(1:是 0:否)
清空后,链表是否空? 1(1:是 0:否)
```

# 第3章 栈和队列

## 3.1 栈

```
// c3-1.h 栈的顺序存储结构。在教科书第 46 页(见图 3-1)
#define STACK_INIT_SIZE 10 // 存储空间初始分配量
#define STACK_INCREMENT 2 // 存储空间分配增量
struct SqStack // 顺序栈
{ SElemType * base; // 在栈构造之前和销毁之后,base 的值为 NULL
  SElemType * top; // 栈顶指针
  int stacksize; // 当前已分配的存储空间,以元素为单位
};
```

SqStack | stacksize | top | base | SElemType | SElemType

图 3-1 顺序栈存储结构

```
// bo3-1.cpp 顺序栈(存储结构由 c3-1.h 定义)的基本操作(9 个)
void InitStack(SqStack &S)
{ // 构造一个空栈 S。在教科书第 47 页(见图 3-2)
  S.base = (SElemType * )malloc(STACK_INIT_SIZE * sizeof(SElemType));
  if(!S.base)
    exit(OVERFLOW); // 动态分配存储空间失败,则退出
  S.top = S.base; // 栈顶指向栈底(空栈)
  S.stacksize = STACK_INIT_SIZE; // 存储空间为初始分配量
}
void DestroyStack(SqStack &S)
{ // 销毁栈 S,S 不再存在(见图 3-3)
  free(S.base); // 释放栈空间
  S.top = S.base = NULL; // 栈顶、栈底指针均为空
  S.stacksize = 0; // 当前已分配的存储空间为 0
}
void ClearStack(SqStack &S)
{ // 把栈 S 置为空栈
  S.top = S.base; // 栈顶指针指向栈底
}
Status StackEmpty(SqStack S)
{ // 若栈 S 为空栈,则返回 TRUE;否则返回 FALSE
```

10 | S | [9] | [8] | ⋮ | [1] | [0]

图 3-2 构造一个空的顺序栈 S

0 | NULL | NULL | S

图 3-3 销毁顺序栈 S

```
  if(S.top == S.base) // 空栈条件
    return TRUE;
  else
    return FALSE;
}
int StackLength(SqStack S)
{ // 返回栈 S 的元素个数,即栈的长度
  return S.top - S.base;
}
Status GetTop(SqStack S,SElemType &e) // 在教科书第 47 页
{ // 若栈 S 不空,则用 e 返回 S 的栈顶元素,并返回 OK;否则返回 ERROR
  if(S.top>S.base) // 栈不空
  { e = *(S.top - 1); // 将栈顶元素赋给 e
    return OK;
  }
  else
    return ERROR;
}
void Push(SqStack &S,SElemType e)
{ // 插入元素 e 为栈 S 新的栈顶元素。
  // 在教科书第 47 页(见图 3-4)
  if(S.top - S.base == S.stacksize) // 栈满
  { S.base = (SElemType *)realloc(S.base,
    (S.stacksize + STACK_INCREMENT) *
    sizeof(SElemType)); // 追加存储空间
    if(!S.base) // 追加存储空间失败,则退出
      exit(OVERFLOW);
    S.top = S.base + S.stacksize; // 修改栈顶指针,指向新的栈顶
    S.stacksize += STACK_INCREMENT; // 更新当前已分配的存储空间
  }
  *(S.top) ++= e; // 将 e 入栈,成为新的栈顶元素,栈顶指针上移 1 个存储单元
}
Status Pop(SqStack &S,SElemType &e) // 在教科书第 47 页
{ // 若栈 S 不空,则删除 S 的栈顶元素,用 e 返回其值,
  // 并返回 OK;否则返回 ERROR(见图 3-5)
  if(S.top == S.base) // 栈空
    return ERROR;
  e = * --S.top; // 将栈顶元素赋给 e,栈顶指针下移
                 // 1 个存储单元
  return OK;
}
void StackTraverse(SqStack S,void( * visit)(SElemType))
{ // 从栈底到栈顶依次对栈 S 中每个元素调用函数 visit()
```


(a) 调用Push()之前　(b) 调用Push()之后

图 3-4　调用 Push()示例


(a) 调用Pop()之前　(b) 调用Pop()之后

图 3-5　调用 Pop()示例

```
  while(S.top>S.base) // S.base 指向栈元素
    visit(*S.base++); // 对该栈元素调用 visit(),
  printf("\n");          // 栈底指针上移 1 个存储单元
}
```

```
// main3-1.cpp 检验 bo3-1.cpp 的主程序
#include"c1.h"
typedef int SElemType; // 定义栈元素类型,此句要在 c3-1.h 的前面
#include"c3-1.h" // 栈的顺序存储结构
#include"bo3-1.cpp" // 顺序栈存储结构的基本操作(9 个)
#define ElemType SElemType // 将 func2-2.cpp 中的 ElemType 类型定义为 SElemType 类型
#include"func2-2.cpp" // 包括 equal()、comp()、print()、print1()和 print2()函数
void main()
{
  int j;
  SqStack s;
  SElemType e;
  InitStack(s); // 初始化栈 s
  for(j=1;j<=12;j++)
    Push(s,j); // 将值为 j 的栈元素入栈 s 中
  printf("栈中元素依次为");
  StackTraverse(s,print); // 从栈底到栈顶依次对栈 s 中每个元素调用 print()函数
  Pop(s,e); // 弹出栈顶元素,其值赋给 e
  printf("弹出的栈顶元素 e=%d\n",e);
  printf("栈空否?%d(1:空 0:否),",StackEmpty(s));
  GetTop(s,e); // 将新的栈顶元素赋给 e
  printf("栈顶元素 e=%d,栈的长度为%d\n",e,StackLength(s));
  ClearStack(s); // 清空栈 s
  printf("清空栈后,栈空否?%d(1:空 0:否)\n",StackEmpty(s));
  DestroyStack(s); // 销毁栈 s
  printf("销毁栈后,s.top=%u,s.base=%u,s.stacksize=%d\n",s.top,s.base,s.stacksize);
}
```

**程序运行结果:**

```
栈中元素依次为 1 2 3 4 5 6 7 8 9 10 11 12
弹出的栈顶元素 e=12
栈空否? 0(1:空 0:否),栈顶元素 e=11,栈的长度为 11
清空栈后,栈空否? 1(1:空 0:否)
销毁栈后,s.top=0,s.base=0,s.stacksize=0
```

# 3.2 栈的应用举例

## 3.2.1 数制转换

```
// algo3-1.cpp 调用算法 3.1 的程序
#define N 8 // 定义待转换的进制 N(2～9)
typedef int SElemType; // 定义栈元素类型为整型
#include"c1.h"
#include"c3-1.h" // 采用顺序栈
#include"bo3-1.cpp" // 利用顺序栈的基本操作
void conversion() // 算法 3.1
{ // 对于输入的任意一个非负十进制整数,打印输出与其等值的 N 进制数
  SqStack s;
  unsigned n; // 非负整数
  SElemType e;
  InitStack(s); // 初始化栈
  printf("将十进制整数 n 转换为 %d 进制数,请输入:n(≥0) = ",N);
  scanf("%u",&n); // 输入非负十进制整数 n
  while(n) // 当 n 不等于 0
  { Push(s,n%N); // 入栈 n 除以 N 的余数(N 进制的低位)
    n = n/N;
  }
  while(!StackEmpty(s)) // 当栈不空
  { Pop(s,e); // 弹出栈顶元素且赋值给 e
    printf("%d",e); // 输出 e
  }
  printf("\n");
}
void main()
{
  conversion();
}
```

| | | |
|---|---|---|
| 10 | | [9] |
| | | [8] |
| | 2 | ⋮ |
| | 5 | |
| | 0 | [1] |
| | 4 | [0] |

s

图 3-6 栈 s 在元素最多时的状态

**程序运行结果**(见图 3-6):

```
将十进制整数 n 转换为 8 进制数,请输入:n(≥0) = 1348↙
2504
```

如果将 N 定义为 2,algo3-1.cpp 就是将十进制数转换为二进制数的程序。

**程序运行结果**:

```
将十进制整数 n 转换为 2 进制数,请输入:n(≥0) = 13↙
1101
```

algo3-1.cpp 能不能用于十进制到十六进制的转换呢?存在一个问题,要将余数 10～

15 转换为 A～F 输出。algo3-2.cpp 实现了十进制到十六进制的转换。

```
// algo3-2.cpp 修改算法 3.1,十进制→十六进制
typedef int SElemType; // 定义栈元素类型为整型
#include"c1.h"
#include"c3-1.h" // 采用顺序栈
#include"bo3-1.cpp" // 利用顺序栈的基本操作
void conversion()
{ // 对于输入的任意一个非负十进制整数,打印输出与其等值的十六进制数
  SqStack s;
  unsigned n; // 非负整数
  SElemType e;
  InitStack(s); // 初始化栈
  printf("将十进制整数 n 转换为十六进制数,请输入:n(≥0) = ");
  scanf("%u",&n); // 输入非负十进制整数 n
  while(n) // 当 n 不等于 0
  { Push(s,n%16); // 入栈 n 除以 16 的余数(十六进制的低位)
    n = n/16;
  }
  while(!StackEmpty(s)) // 当栈不空
  { Pop(s,e); // 弹出栈顶元素且赋值给 e
    if(e<=9)
      printf("%d",e); // 小于等于 9 的余数,直接输出
    else
      printf("%c",e+55); // 大于 9 的余数,输出相应的字符
  }
  printf("\n");
}
void main()
{
  conversion();
}
```

**程序运行结果**(见图 3-7):

```
将十进制整数 n 转换为十六进制数,请输入:n(≥0) = 164↙
A4
```

[9]
[8]
⋮
10 [1]
4 [0]
10
s

图 3-7　栈 s 在元素最多时的状态

## 3.2.2　行编辑程序

```
// algo3-3.cpp 行编辑程序,实现算法 3.2
typedef char SElemType; // 定义栈元素类型为字符型
#include"c1.h"
#include"c3-1.h" // 栈的顺序存储结构
```

```
#include"bo3-1.cpp" // 顺序栈存储结构的基本操作(9 个)
FILE *fp;
void copy(SElemType c)
{ // 将字符 c 送至 fp 所指的文件中
  fputc(c,fp);
}
void LineEdit()
{ // 利用字符栈 s,从终端接收一行并送至调用过程的数据区。算法 3.2
  SqStack s;
  char ch;
  InitStack(s); // 初始化栈 s
  printf("请输入一个文本文件,^Z 结束输入:\n");
  ch=getchar(); // 接收字符到 ch
  while(ch!=EOF) // 当全文未结束(EOF 为^Z 键,全文结束符)
  { while(ch!=EOF&&ch!='\n') // 当全文未结束且未到行末(不是换行符)
      { switch(ch) // 对于当前字符 ch,分情况处理
      { case '#':if (!StackEmpty(s))
                    Pop(s,ch); // 仅当栈非空时弹出栈顶元素,c 可由 ch 替代
                  break;
        case '@':ClearStack(s); // 重置 s 为空栈
                  break;
        default :Push(s,ch); // 其他字符进栈
      }
      ch=getchar(); // 从终端接收下一个字符到 ch
    } // 到行末或全文结束,退出此层循环
    StackTraverse(s,copy); // 将从栈底到栈顶的栈内字符依次传送至文件(调用 copy()函数)
    fputc('\n',fp); // 向文件输入一个换行符
    ClearStack(s); // 重置 s 为空栈
    if(ch!=EOF) // 全文未结束
      ch=getchar(); // 从终端接收下一个字符到 ch
  }
  DestroyStack(s); // 销毁栈 s
}
void main()
{
  fp=fopen("ed.txt","w");
  // 在当前目录下建立 ed.txt 文件,用于写数据,如已有同名文件则先删除原文件
  if(fp) // 建立文件成功
  { LineEdit(); // 行编辑
    fclose(fp); // 关闭 fp 所指的文件
  }
  else
    printf("建立文件失败!\n");
}
```

**程序运行结果**(以教科书第 49 页的输入为例)：

```
请输入一个文本文件,^Z 结束输入：
whli##ilr#e(s#*s)↙

  outcha@putchar(*s=#++);↙

^Z↙
```

文件 ed.txt 的内容：

```
while(*s)
putchar(*s++);
```

### 3.2.3 迷宫求解

```
// algo3-4.cpp 利用栈求解迷宫问题(只输出一个解,算法 3.3)
#include"c1.h"
struct PosType // 迷宫坐标位置类型(见图 3-8)
{ int x; // 行值
  int y; // 列值
};
// 全局变量
PosType begin,end; // 迷宫的入口坐标,出口坐标
PosType direc[4]={{0,1},{1,0},{0,-1},{-1,0}};
// {行增量,列增量},移动方向依次为东南西北
#define MAXLENGTH 25 // 设迷宫的最大行列为 25
typedef int MazeType[MAXLENGTH][MAXLENGTH]; // 迷宫数组类型[行][列]
MazeType m; // 迷宫数组
int x,y; // 迷宫的行数,列数
void Print()
{ // 输出迷宫的解(m 数组)
  int i,j;
  for(i=0;i<x;i++)
  { for(j=0;j<y;j++)
      printf("%3d",m[i][j]);
    printf("\n");
  }
}
void Init()
{ // 设定迷宫布局(墙值为 0,通道值为 1)
  int i,j,x1,y1;
  printf("请输入迷宫的行数,列数(包括外墙)：");
  scanf("%d,%d",&x,&y);
```

PosType

| x | y |
|---|---|

图 3-8　PosType 结构

```
  for(i=0;i<y;i++) // 定义周边值为0(外墙)
  { m[0][i]=0; // 上边
    m[x-1][i]=0; // 下边
  }
  for(i=1;i<x-1;i++)
  { m[i][0]=0; // 左边
    m[i][y-1]=0; // 右边
  }
  for(i=1;i<x-1;i++)
    for(j=1;j<y-1;j++)
      m[i][j]=1; // 定义除外墙,其余都是通道,初值为1
  printf("请输入迷宫内墙单元数:");
  scanf("%d",&j);
  printf("请依次输入迷宫内墙每个单元的行数,列数:\n");
  for(i=1;i<=j;i++)
  { scanf("%d,%d",&x1,&y1);
    m[x1][y1]=0; // 修改内墙的值为0
  }
  printf("迷宫结构如下:\n");
  Print();
  printf("请输入入口的行数,列数:");
  scanf("%d,%d",&begin.x,&begin.y);
  printf("请输入出口的行数,列数:");
  scanf("%d,%d",&end.x,&end.y);
}
int curstep=1; // 当前足迹,初值(在入口处)为1
struct SElemType // 栈的元素类型。在教科书第51页(见图3-9)
{ int ord; // 通道块在路径上的"序号"
  PosType seat; // 通道块在迷宫中的"坐标位置"
  int di; // 从此通道块走向下一通道块的"方向"(0~3表示东~北)
};
```

SElemType

| ord | seat.x | seat.y | di |
|---|---|---|---|

图 3-9　SElemType 结构

```
#include"c3-1.h" // 采用顺序栈存储结构
#include"bo3-1.cpp" // 采用顺序栈的基本操作函数
// 定义墙元素值为0,可通过路径为1,经试探不可通过路径为-1,通过路径为足迹
Status Pass(PosType b)
{ // 当迷宫m的b点的值为1(可通过路径),返回OK;否则,返回ERROR
  if(m[b.x][b.y]==1)
    return OK;
  else
    return ERROR;
}
void FootPrint(PosType b)
{ // 使迷宫m的b点的值变为足迹(curstep)
  m[b.x][b.y]=curstep;
}
void NextPos(PosType &b,int di)
{ // 根据当前位置b及移动方向di,修改b为下一位置
```

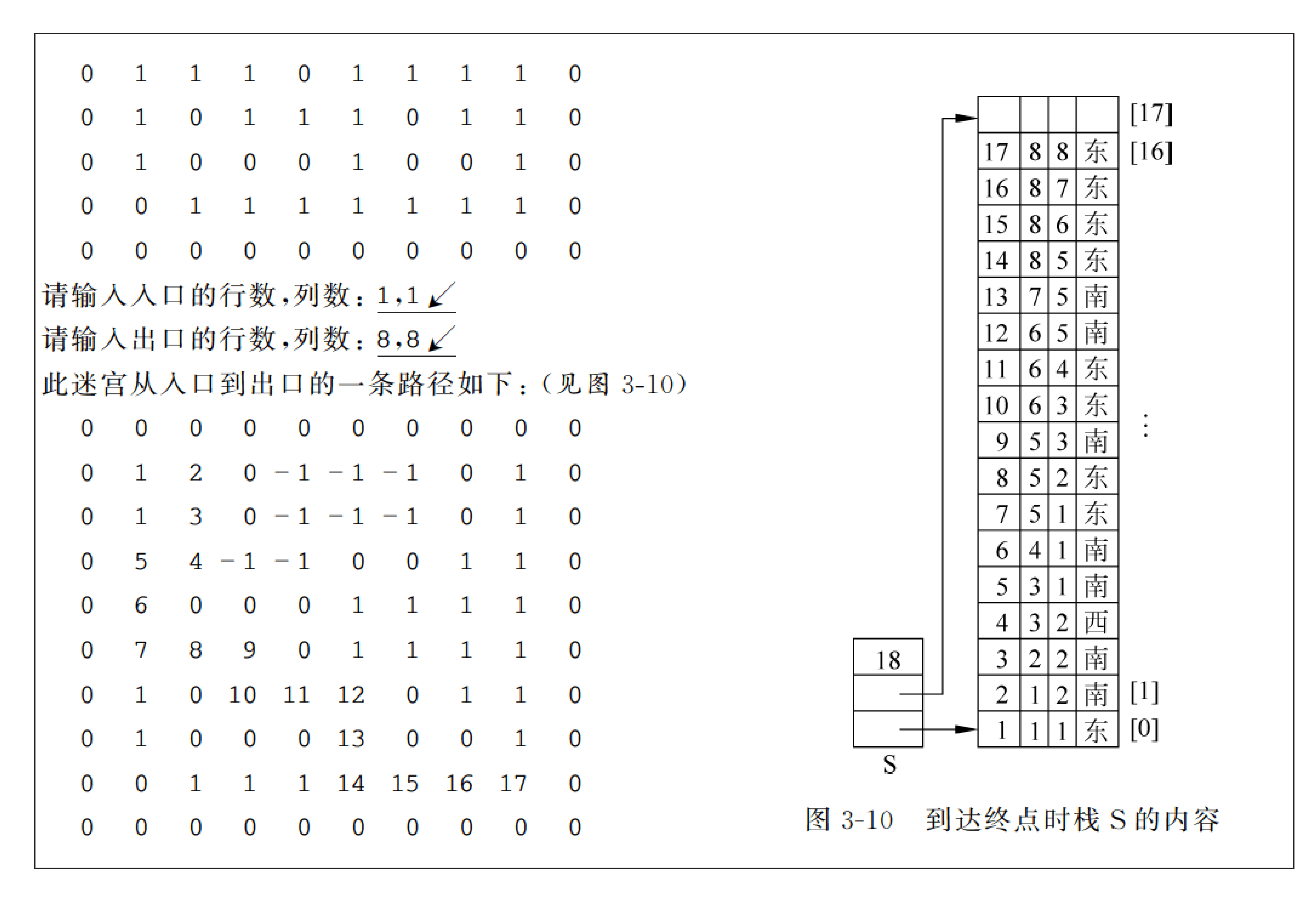
```
    b.x += direc[di].x;
    b.y += direc[di].y;
  }
  void MarkPrint(PosType b)
  { // 使迷宫 m 的 b 点的值变为 -1(标记为经试探由此不能到达终点的路径)
    m[b.x][b.y] = -1;
  }
  Status MazePath(PosType start,PosType end) // 算法 3.3
  { // 若迷宫 m 中存在从入口 start 到出口 end 的通道,则求得一条存放在栈中(从栈底到栈顶),
    // 并返回 TRUE;否则返回 FALSE
    PosType curpos = start; // 当前位置在入口
    SqStack S; // 顺序栈
    SElemType e; // 栈元素
    InitStack(S); // 初始化栈
    do
    { if(Pass(curpos)) // 当前位置可以通过,即是未曾走到过的通道块
      { FootPrint(curpos); // 留下足迹(使迷宫 m 的当前位置的值为 curstep)
        e.ord = curstep; // 栈元素的序号为当前足迹 curstep
        e.seat = curpos; // 栈元素的位置为当前位置
        e.di = 0; // 从当前位置出发,下一位置是向东
        Push(S,e); // 入栈当前位置及状态
        curstep ++; // 足迹加 1
        if(curpos.x == end.x&&curpos.y == end.y) // 到达终点(出口)
          return TRUE;
        NextPos(curpos,e.di); // 由当前位置及移动方向,确定下一个当前位置,仍赋给 curpos
      }
      else // 当前位置不能通过
        { if(!StackEmpty(S)) // 栈不空
        { Pop(S,e); // 退栈到前一位置
          curstep --; // 足迹减 1
          while(e.di == 3&&!StackEmpty(S)) // 前一位置处于最后一个方向(北)且栈不空
          { MarkPrint(e.seat); // 在前一位置留下由其不能到达终点的标记(-1)
            Pop(S,e); // 再退回一步
            curstep --; // 足迹再减 1
          }
          if(e.di<3) // 未到最后一个方向(北)
          { e.di ++; // 换下一个方向探索
            Push(S,e); // 入栈该位置的下一个方向
            curstep ++; // 足迹加 1
            curpos = e.seat; // 确定当前位置
            NextPos(curpos,e.di); // 由当前位置及移动方向,确定下一个当前位置,仍赋给 curpos
          }
        }
      }
```

```
  }while(!StackEmpty(S));
  return FALSE;
}
void main()
{
  Init(); // 初始化迷宫
  if(MazePath(begin,end)) // 有通路
  { printf("此迷宫从入口到出口的一条路径如下:\n");
    Print(); // 输出此通路(m 数组)
  }
  else
    printf("此迷宫没有从入口到出口的路径\n");
}
```

由于把迷宫数组 m、迷宫的行数 x、列数 y、迷宫的入口坐标 begin 和出口坐标 end 作为全局变量,它们在各函数中都通用,不需要用形参来传递,减少了 Print()函数和 Init()函数中的形参数量。

**程序运行结果**(以教科书图 3.4 为例):

```
请输入迷宫的行数,列数(包括外墙):10,10↙
请输入迷宫内墙单元数:18↙
请依次输入迷宫内墙每个单元的行数,列数:
1,3↙
1,7↙
2,3↙
2,7↙
3,5↙
3,6↙
4,2↙
4,3↙
4,4↙
5,4↙
6,2↙
6,6↙
7,2↙
7,3↙
7,4↙
7,6↙
7,7↙
8,1↙
迷宫结构如下:
   0   0   0   0   0   0   0   0   0   0
   0   1   1   0   1   1   1   0   1   0
   0   1   1   0   1   1   1   0   1   0
   0   1   1   1   1   0   0   1   1   0
   0   1   0   0   0   1   1   1   1   0
```



```
  0   1   1   1   0   1   1   1   1   0
  0   1   0   1   1   1   0   1   1   0
  0   1   0   0   0   1   0   0   1   0
  0   0   1   1   1   1   1   1   1   0
  0   0   0   0   0   0   0   0   0   0
请输入入口的行数,列数:1,1↙
请输入出口的行数,列数:8,8↙
此迷宫从入口到出口的一条路径如下:(见图 3-10)
  0   0   0   0   0   0   0   0   0   0
  0   1   2   0  -1  -1  -1   0   1   0
  0   1   3   0  -1  -1  -1   0   1   0
  0   5   4  -1  -1   0   0   1   1   0
  0   6   0   0   0   1   1   1   1   0
  0   7   8   9   0   1   1   1   1   0
  0   1   0  10  11  12   0   1   1   0
  0   1   0   0   0  13   0   0   1   0
  0   0   1   1   1  14  15  16  17   0
  0   0   0   0   0   0   0   0   0   0
```



图 3-10　到达终点时栈 S 的内容

从入口到出口的一条路径由全局变量 m 数组显示。入口的值为 1,路径的值依次为 2,3,…,17。图 3-10 显示了到达终点时栈的内容。其中栈元素的 di 成员的值本是 0～3,为直观,用东、南、西、北代替。由于有数组 m,栈的主要作用并不是保存路径。它的主要作用是当试探失败时,通过退栈回到前一点,从前一点再继续试探。

这条路径共走了 16 步,从入口(第 1 步)到第 5 步其实只须走 2 步。原因是 func3-1.cpp 中定义的数组 direc[]的移动方向依次为东、南、西、北。程序由入口处先向东试探。既然有路,就不再向南试探了。因此走了弯路。如果将 direc[0] 和 direc[1]的值交换一下,新的移动方向依次为南、东、西、北。这样,程序先向南试探,只在不成功时才向东试探。避免了在此处走弯路,14 步就到了出口。以下是修改了 direc[]的值为"direc[4]={{1,0},{0,1},{0,-1},{-1,0}};",不改变输入数据的情况下的运行结果。为节约篇幅,略去相同部分,只列出最后结果。

**程序运行结果:**

```
此迷宫从入口到出口的一条路径如下:
  0   0   0   0   0   0   0   0   0   0
  0   1   1   0   1   1   1   0   1   0
  0   2   1   0   1   1   1   0   1   0
  0   3   1   1   1   0   0   1   1   0
  0   4   0   0   0   1   1   1   1   0
  0   5   6   7   0   1   1   1   1   0
  0  -1   0   8   9  10   0   1   1   0
  0  -1   0   0   0  11   0   0   1   0
  0   0   1   1   1  12  13  14  15   0
  0   0   0   0   0   0   0   0   0   0
```

在 m 数组中，－1 表示曾试探过，但不能通行又退回去的路径；除入口以外，1 表示没有走过的路径；大于 1 的值表示由入口通向出口的足迹。上面 m 数组第 5 步的下面有 2 个－1。它们的值本是 1(通道)。当走到第 5 步时，入栈{5,5,1,0}，说明第 5 步走在 5 行 1 列，且下一步是向南走。向南走是通路，将 m 数组 6 行 1 列的值改为 6(留下足迹)，入栈{6,6,1,0}，当前足迹 curstep 加 1 为 7。说明第 6 步走在 6 行 1 列，且下一步也是向南走。还是通路，将 m 数组 7 行 1 列的值改为 7(将 curstep 赋给 m[7][1])，入栈{7,7,1,0}，当前足迹 curstep 加 1 为 8。说明第 7 步走在 7 行 1 列，且下一步也是向南走。这时不再是通路，出栈{7,7,1,0}。改变方向，入栈{7,7,1,1}。说明第 7 步仍是走在 7 行 1 列，下一步改为向东走。依然没有通路。7 行 1 列的 4 个方向上都不是 1(走投无路)。最后，将 m 数组 7 行 1 列的值改为－1，出栈{7,7,1,3}，当前足迹 curstep 减 1 为 6。再出栈{6,6,1,0}。改变方向，入栈{6,6,1,1}。6 行 1 列的 4 个方向上也都不是 1。最后，将 m 数组 6 行 1 列的值改为－1，出栈{6,6,1,3}，当前足迹 curstep 减 1 为 5。再出栈{5,5,1,0}。改变方向，入栈{5,5,1,1}。向第 5 步的东面试探第 6 步，……，直到终点或没有通路。

### 3.2.4　表达式求值

```
// func3-1.cpp algo3-5.cpp 和 algo3-6.cpp 要调用的函数
char Precede(SElemType t1,SElemType t2)
{ // 根据教科书表 3.1,判断 t1,t2 两符号的优先关系('#'用'\n'代替)
  char f;
  switch(t2)
  { case '+':
    case '-':if(t1 == '('||t1 == '\n')
               f = '<'; // t1<t2
             else
               f = '>'; // t1>t2
             break;
    case '*':
    case '/' :if(t1 == '*'||t1 == '/'||t1 == ')')
               f = '>'; // t1>t2
             else
               f = '<'; // t1<t2
             break;
    case '(' :if(t1 == ')')
             { printf("括号不匹配\n");
               exit(OVERFLOW);
             }
             else
               f = '<'; // t1<t2
             break;
    case ')' :switch(t1)
             { case  '(':f = '='; // t1 = t2
```

```
                    break;
                case '\n':printf("缺乏左括号\n");
                    exit(OVERFLOW);
                default :f = '>'; // t1>t2
              }
              break;
    case'\n' :switch(t1)
              { case '\n':f = '='; // t1 = t2
                    break;
                case '(': printf("缺乏右括号\n");
                    exit(OVERFLOW);
                default : f = '>'; // t1>t2
              }
  }
  return f;
}
Status In(SElemType c)
{ // 判断 c 是否为 7 种运算符之一
  switch(c)
  { case '+':
    case '-':
    case '*':
    case '/' :
    case '(' :
    case ')' :
    case'\n' :return TRUE;
    default :return FALSE;
  }
}
SElemType Operate(SElemType a,SElemType theta,SElemType b)
{ // 做四则运算 a theta b,返回运算结果
  switch(theta)
  { case'+':return a + b;
    case'-':return a - b;
    case'*':return a * b;
  }
  return a/b;
}
```

```
// algo3-5.cpp 表达式求值(输入的值在 0～9 之间,中间结果和输出的值在 - 128～127 之间),
// 算法 3.4
typedef char SElemType; // 定义栈元素为字符型
#include"c1.h"
#include"c3-1.h" // 栈的顺序存储结构
```

```
#include"bo3-1.cpp" // 顺序栈存储结构的基本操作(9 个)
#include"func3-1.cpp" // 包括 Precede()、In()和 Operate()函数
SElemType EvaluateExpression() // 算法 3.4,有改动
{ // 算术表达式求值的算符优先算法。设 OPTR 和 OPND 分别为运算符栈和运算数栈
  SqStack OPTR,OPND;
  SElemType a,b,c,x;
  InitStack(OPTR); // 初始化运算符栈 OPTR 和运算数栈 OPND
  InitStack(OPND);
  Push(OPTR,'\n'); // 将换行符压入运算符栈 OPTR 的栈底。修改
  c=getchar(); // 由键盘读入 1 个字符到 c
  GetTop(OPTR,x); // 将运算符栈 OPTR 的栈顶元素赋给 x
  while(c!='\n'||x!='\n') // c 和 x 不都是换行符
  { if(In(c)) // c 是 7 种运算符之一
      switch(Precede(x,c)) // 判断 x 和 c 的优先权
      { case'<':Push(OPTR,c); // 运算符栈 OPTR 的栈顶元素 x 的优先权低,入栈 c
                c=getchar(); // 由键盘读入下一个字符到 c
                break;
        case'=':Pop(OPTR,x); // x='('且 c=')'情况,弹出'('给 x(后又扔掉)
                c=getchar(); // 由键盘读入下一个字符到 c(扔掉 c 原有的')')
                break;
        case'>':Pop(OPTR,x); // 栈顶元素 x 的优先权高,弹出运算符栈 OPTR 的栈顶元素给 x。修改
                Pop(OPND,b); // 依次弹出运算数栈 OPND 的栈顶元素给 b,a
                Pop(OPND,a);
                Push(OPND,Operate(a,x,b)); // 做运算 a x b,并将运算结果入运算数栈
      }
    else if(c>='0'&&c<='9') // c 是操作数
    { Push(OPND,c-48); // 将该操作数的值(不是 ASCII 码)压入运算数栈 OPND
      c=getchar(); // 由键盘读入下一个字符到 c
    }
    else // c 是非法字符
    { printf("出现非法字符\n");
      exit(OVERFLOW);
    }
    GetTop(OPTR,x); // 将运算符栈 OPTR 的栈顶元素赋给 x
  }
  Pop(OPND,x); // 弹出运算数栈 OPND 的栈顶元素(运算结果)给 x(修改此处)
  if(!StackEmpty(OPND)) // 运算数栈 OPND 不空(运算符栈 OPTR 仅剩'\n')
  { printf("表达式不正确\n");
    exit(OVERFLOW);
  }
  return x;
}
void main()
{
  printf("请输入算术表达式(输入的值要在 0~9 之间、");
  printf("中间运算值和输出结果在-128~127 之间)\n");
```

```
  printf("%d\n",EvaluateExpression()); // 返回值(8 位二进制,1 个字节)按整型格式输出
}
```

**程序运行结果**(以教科书例 3-1 为例):

```
请输入算术表达式(输入的值要在 0～9 之间、中间运算值和输出结果在 -128～127 之间)
3*(7-2)↙
15
```

algo3-6.cpp 对 algo3-5.cpp 做了些改进,把连续输入的几个数值型字符作为一个整数处理。使数值范围扩大为整型,更具有实用性。因为只包括四则运算符,故负数要用(0—正数)表示。

```
// algo3-6.cpp 表达式求值(范围为 int 类型,输入负数要用(0 - 正数)表示)
typedef int SElemType; // 定义栈元素类型为整型,修改 algo3-5.cpp
#include"c1.h"
#include"c3-1.h" // 顺序栈的存储结构
#include"bo3-1.cpp" // 顺序栈的基本操作
#include"func3-1.cpp" // 包括 Precede()、In()和 Operate()函数
SElemType EvaluateExpression()
{ // 算术表达式求值的算符优先算法。设 OPTR 和 OPND 分别为运算符栈和运算数栈
  SqStack OPTR,OPND;
  SElemType a,b,d,x; // 修改 algo3-5.cpp
  char c; // 存放由键盘接收的字符,修改 algo3-5.cpp
  InitStack(OPTR); // 初始化运算符栈 OPTR 和运算数栈 OPND
  InitStack(OPND);
  Push(OPTR,'\n'); // 将换行符压入运算符栈 OPTR 的栈底。修改
  c=getchar(); // 由键盘读入 1 个字符到 c
  GetTop(OPTR,x); // 将运算符栈 OPTR 的栈顶元素赋给 x
  while(c!='\n'||x!='\n') // c 和 x 不都是换行符
  { if(In(c)) // c 是 7 种运算符之一
      switch(Precede(x,c)) // 判断 x 和 c 的优先权
      { case'<':Push(OPTR,c); // 运算符栈 OPTR 的栈顶元素 x 的优先权低,入栈 c
                c=getchar(); // 由键盘读入下一个字符到 c
                break;
        case'=':Pop(OPTR,x); // x='('且 c=')'情况,弹出'('给 x(后又扔掉)
                c=getchar(); // 由键盘读入下一个字符到 c(扔掉 c 原有的')')
                break;
        case'>':Pop(OPTR,x); // 栈顶元素 x 的优先权高,弹出运算符栈 OPTR 的栈顶元素给 x。修改
               Pop(OPND,b); // 依次弹出运算数栈 OPND 的栈顶元素给 b,a
               Pop(OPND,a);
               Push(OPND,Operate(a,x,b)); // 做运算 a x b,并将运算结果入运算数栈
      }
    else if(c>='0'&&c<='9') // c 是操作数,此语句修改 algo3-5.cpp
    { d=0;
```

```
      while(c>='0'&&c<='9') // 是连续数字
      { d=d*10+c-'0';
        c=getchar();
      }
      Push(OPND,d); // 将 d 压入运算数栈 OPND
    }
    else // c 是非法字符,以下同 algo3-5.cpp
    { printf("出现非法字符\n");
      exit(OVERFLOW);
    }
    GetTop(OPTR,x); // 将运算符栈 OPTR 的栈顶元素赋给 x
  }
  Pop(OPND,x); // 弹出运算数栈 OPND 的栈顶元素(运算结果)给 x(修改此处)
  if(!StackEmpty(OPND)) // 运算数栈 OPND 不空(运算符栈 OPTR 仅剩'\n')
  { printf("表达式不正确\n");
    exit(OVERFLOW);
  }
  return x;
}
void main()
{
  printf("请输入算术表达式,负数要用(0-正数)表示\n");
  printf("%d\n",EvaluateExpression());
}
```

**程序运行结果:**

```
请输入算术表达式,负数要用(0-正数)表示
(0-12)*((5-3)*3+2)/(2+2)↙
-24
```

## 3.3 栈与递归的实现

函数中有直接或间接地调用自身函数的语句,这样的函数称为递归函数。递归函数用得好,可简化编程工作。但函数自己调用自己,有可能造成死循环。为了避免死循环,要做到两点:

(1) 降阶。递归函数虽然调用自身,但并不是简单地重复。它的实参值每次是不一样的。一般逐渐减小,称为降阶。如教科书式(3-3)的 Ackerman 函数,当 m≠0 时,求 Ack(m,n)可由 Ack(m−1,…)得到,Ack()函数的第 1 个参数减小了。

(2) 有出口。即在某种条件下,不再进行递归调用。仍以教科书式(3-3)的 Ackerman 函数为例,当 m=0 时,Ack(0,n)=n+1,终止了递归调用。所以,递归函数总有条件语句。m=0 的条件是由逐渐降阶形成的。如取 Ack(m,n)函数的实参 m=−1,即使通过降阶也

不会出现 m＝0 的情况，这也会造成死循环。

系统在遇到函数调用时，会将调用函数的实参、返回地址等信息存入系统自身设立的“递归工作栈”中，再去运行被调函数。从被调函数返回时，再将调用函数的信息出栈，接着运行调用函数。在一系列递归调用过程中，最后递归调用的函数最先结束调用返回主调函数。所以把它们的信息存入栈中是很合适的。系统开辟的栈空间是有限的，当递归调用时，嵌套的层次往往很多，就有可能使栈发生溢出现象，从而出现不可预料的后果。运行 algo3-7.cpp 时，m、n 的取值就不可过大。

```
// algo3-7.cpp 用递归调用求 Ackerman(m,n)的值
#include<stdio.h>
int ack(int m,int n)
{ int z;
  if(m == 0)
    z = n + 1; // 出口
  else if(n == 0)
    z = ack(m - 1,1); // 对形参 m 降阶
  else
    z = ack(m - 1,ack(m,n - 1)); // 对形参 m、n 降阶
  return z;
}
void main()
{
  int m,n;
  printf("请输入 m,n: ");
  scanf("%d,%d",&m,&n);
  printf("Ack(%d,%d) = %d\n",m,n,ack(m,n));
}
```

**程序运行结果**(m、n 不可取值过大)：

```
请输入 m,n: 3,9↙
Ack(3,9) = 4093
```

```
// algo3-8.cpp Hanoi 塔问题，调用算法 3.5 的程序
#include<stdio.h>
int c = 0; // 全局变量，搬动次数
void move(char x,int n,char z)
{ // 第 n 个圆盘从塔座 x 搬到塔座 z
  printf("第 %i 步: 将 %i 号盘从 %c 移到 %c\n",++c,n,x,z);
}
void hanoi(int n,char x,char y,char z) // 算法 3.5
{ // 将塔座 x 上按直径由小到大且自上而下编号为 1 至 n 的 n 个圆盘
  // 按规则搬到塔座 z 上。y 可用作辅助塔座
  if(n == 1) // (出口)
    move(x,1,z); // 将编号为 1 的圆盘从 x 移到 z
```

```
  else
  { hanoi(n-1,x,z,y); // 将 x 上编号为 1 至 n-1 的圆盘移到 y,z 作辅助塔(降阶递归调用)
    move(x,n,z); // 将编号为 n 的圆盘从 x 移到 z
    hanoi(n-1,y,x,z); // 将 y 上编号为 1 至 n-1 的圆盘移到 z,x 作辅助塔(降阶递归调用)
  }
}
void main()
{
  int n;
  printf("3 个塔座为 a、b、c,圆盘最初在 a 座,借助 b 座移到 c 座。请输入圆盘数:");
  scanf("%d",&n);
  hanoi(n,'a','b','c');
}
```

**程序运行结果:**

```
3 个塔座为 a、b、c,圆盘最初在 a 座,借助 b 座移到 c 座。请输入圆盘数: 3↙
第 1 步: 将 1 号盘从 a 移到 c
第 2 步: 将 2 号盘从 a 移到 b
第 3 步: 将 1 号盘从 c 移到 b
第 4 步: 将 3 号盘从 a 移到 c
第 5 步: 将 1 号盘从 b 移到 a
第 6 步: 将 2 号盘从 b 移到 c
第 7 步: 将 1 号盘从 a 移到 c
```

# 3.4　队　　列

## 3.4.1　链队列——队列的链式表示和实现

```
// c3-2.h 单链队列 - 队列的链式存储结构。在教科书第 61 页
typedef struct QNode // (见图 3-11)
{   QElemType data;
    QNode * next;
} * QueuePtr;
struct LinkQueue // (见图 3-12)
{ QueuePtr front,rear; // 队头、队尾指针
};
```

和栈一样,队列也是操作受限的线性表,只允许在队尾插入元素,在队头删除元素。对于链队列结构,为了便于插入元素,设立了队尾指针。这样,插入元素的操作与队列长度无关。图 3-13 是具有 2 个元素的链队列示例。

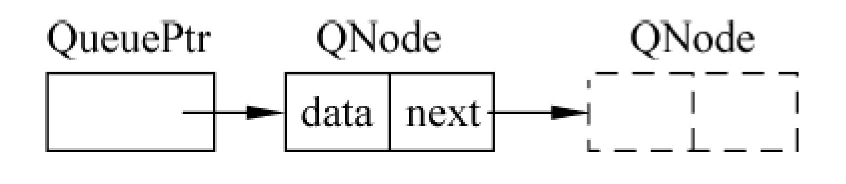


图 3-11　单链队列的结点类型

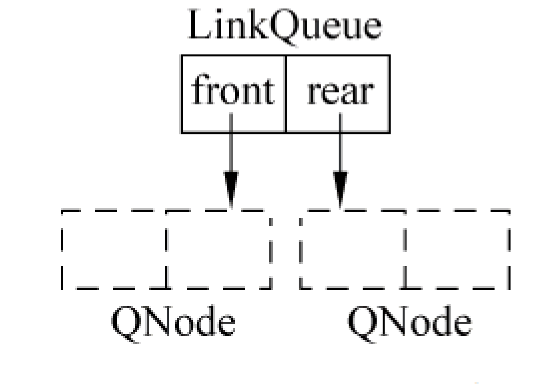


图 3-12　LinkQueue 类型

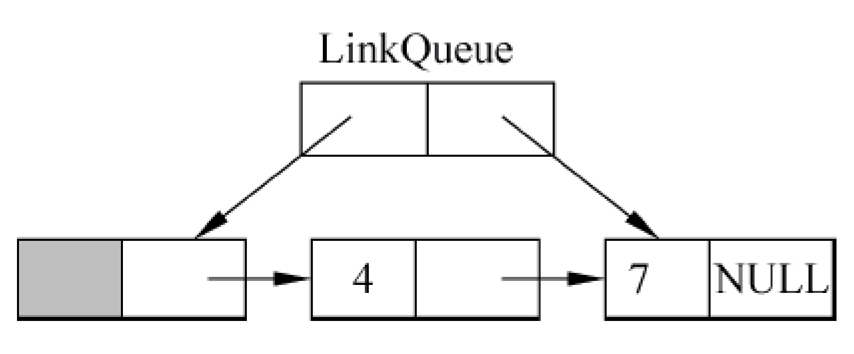


图 3-13　具有 2 个元素(4,7)的链队列

```
// bo3-2.cpp 链队列(存储结构由 c3-2.h 定义)的基本操作(9 个)
void InitQueue(LinkQueue &Q)
{ // 构造一个空队列 Q。在教科书第 62 页(见图 3-14)
  Q.front = Q.rear = (QueuePtr)malloc(sizeof(QNode)); // 生成头结点
  if(!Q.front) // 生成头结点失败
    exit(OVERFLOW);
  Q.front->next = NULL; // 头结点的 next 域为空
}
void DestroyQueue(LinkQueue &Q)
{ // 销毁队列 Q(无论空否均可)。在教科书第 62 页(见图 3-15)
  while(Q.front) // Q.front 不为空
  { Q.rear = Q.front->next; // Q.rear 指向 Q.front 的下一个结点
    free(Q.front); // 释放 Q.front 所指结点
    Q.front = Q.rear; // Q.front 指向 Q.front 的下一个结点
  }
}
void ClearQueue(LinkQueue &Q)
{ // 将队列 Q 清为空队列(见图 3-14)
  DestroyQueue(Q); // 销毁队列 Q(见图 3-15)
  InitQueue(Q); // 重新构造空队列 Q
}
Status QueueEmpty(LinkQueue Q)
{ // 若队列 Q 为空队列,则返回 TRUE;否则返回 FALSE
  if(Q.front->next == NULL)
    return TRUE;
  else
    return FALSE;
}
int QueueLength(LinkQueue Q)
{ // 求队列 Q 的长度
  int i = 0; // 计数器,初值为 0
  QueuePtr p = Q.front; // p 指向头结点
  while(Q.rear != p) // p 所指不是尾结点
  { i++; // 计数器 + 1
    p = p->next; // p 指向下一个结点
  }
  return i;
}
Status GetHead(LinkQueue Q,QElemType &e)
{ // 若队列 Q 不空,则用 e 返回 Q 的队头元素,并返回 OK;否则返回 ERROR
  QueuePtr p;
  if(Q.front == Q.rear) // 队列空
    return ERROR;
  p = Q.front->next; // p 指向队头结点
```

Q

NULL

图 3-14 空队列 Q

Q

NULL NULL

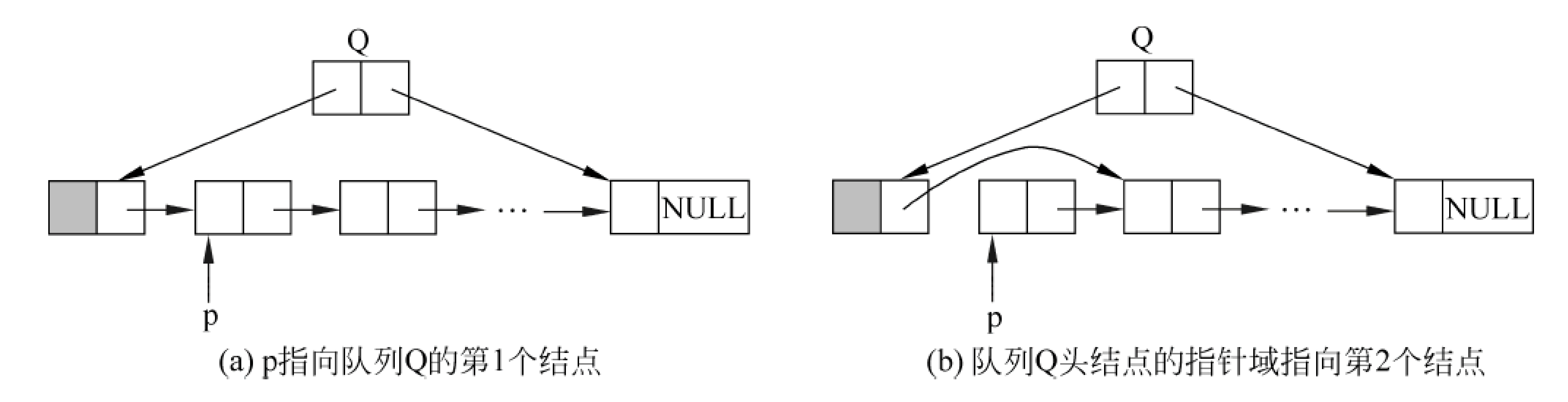
图 3-15 销毁后的队列 Q

```
  e = p->data; // 将队头元素的值赋给 e
  return OK;
}
void EnQueue(LinkQueue &Q,QElemType e)
{ // 插入元素 e 为队列 Q 的新的队尾元素。在教科书第 62 页(见图 3-16)
  QueuePtr p;
  p = (QueuePtr)malloc(sizeof(QNode)); // 动态生成新结点
  if(!p)
    exit(OVERFLOW); // 失败则退出
  p->data = e; // 将值 e 赋给新结点
  p->next = NULL; // 新结点的指针域为空
  Q.rear->next = p; // 原队尾结点的指针指向新结点
  Q.rear = p; // 尾指针指向新结点
}
```

(a) p生成队尾结点　　(b) 将队尾结点插入队列

图 3-16　在队列 Q 的尾部插入元素 e

```
Status DeQueue(LinkQueue &Q,QElemType &e)
{ // 若队列 Q 不空,删除 Q 的队头元素,用 e 返回其值,
  // 并返回 OK;否则返回 ERROR。在教科书第 62 页(见图 3-17)
```




(a) p指向队列Q的第1个结点　　(b) 队列Q头结点的指针域指向第2个结点

图 3-17　删除队列 Q 的第 1 个元素

```
  QueuePtr p;
  if(Q.front == Q.rear) // 队列空
    return ERROR;
  p = Q.front->next; // p 指向队头结点
  e = p->data; // 将队头元素的值赋给 e
  Q.front->next = p->next; // 头结点指向下一个结点
  if(Q.rear == p) // 删除的是队尾结点
    Q.rear = Q.front; // 修改队尾指针指向头结点(空队列)
  free(p); // 释放队头结点
  return OK;
}
```

```
void QueueTraverse(LinkQueue Q,void(* visit)(QElemType))
{ // 从队头到队尾依次对队列 Q 中每个元素调用函数 visit()
  QueuePtr p = Q.front -> next; // p 指向队头结点
  while(p) // p 指向结点
  { visit(p -> data); // 对 p 所指元素调用 visit()
    p = p -> next; // p 指向下一个结点
  }
  printf("\n");
}

// func3-2.cpp 链队列的主函数,main3-2.cpp 和 main3-3.cpp 调用
void main()
{
  int i;
  QElemType d;
  LinkQueue q;
  InitQueue(q); // 构造空队列 q,失败则退出
  printf("成功地构造了一个空队列\n");
  printf("是否空队列? %d(1:空 0:否),",QueueEmpty(q));
  printf("队列的长度为 %d\n",QueueLength(q));
  EnQueue(q, - 5); // 依次入队 3 个元素
  EnQueue(q,5);
  EnQueue(q,10);
  printf("插入 3 个元素( - 5,5,10)后,队列的长度为 %d\n",QueueLength(q));
  printf("是否空队列? %d(1:空 0:否),",QueueEmpty(q));
  printf("队列的元素依次为");
  QueueTraverse(q,print); // 从队头到队尾依次对队列 q 中每个元素调用函数 print()
  i = GetHead(q,d); // 将队头元素赋给 d
  if(i == OK) // 队列 q 不空
    printf("队头元素是 %d,",d);
  DeQueue(q,d); // 删除队头元素,其值赋给 d
  printf("删除了队头元素 %d,",d);
  i = GetHead(q,d); // 将队列 q 的队头元素赋给 d
  if(i == OK) // 队列 q 不空
    printf("新的队头元素是 %d\n",d);
  ClearQueue(q); // 清空队列 q
  printf("清空队列后,q.front = %u,q.rear = %u,q.front -> next = %u\n",q.front,
  q.rear,q.front -> next);
  DestroyQueue(q); // 销毁队列 q
  printf("销毁队列后,q.front = %u,q.rear = %u\n",q.front,q.rear);
}

// main3-2.cpp 检验 bo3-2.cpp 的主程序
#include"c1.h"
```

```
typedef int QElemType; // 定义队列元素为整型
#include"c3-2.h" // 队列的链式存储结构
#include"bo3-2.cpp" // 链队列存储结构的基本操作(9个)
#define ElemType QElemType // 将 func2-2.cpp 中的 ElemType 类型定义为 QElemType 类型
#include"func2-2.cpp" // 包括 equal()、comp()、print()、print1()和 print2()函数
#include"func3-2.cpp" // 主函数
```

**程序运行结果：**

```
成功地构造了一个空队列
是否空队列？1(1:空 0:否),队列的长度为 0
插入 3 个元素(-5,5,10)后,队列的长度为 3
是否空队列？0(1:空 0:否),队列的元素依次为 -5 5 10
队头元素是 -5,删除了队头元素 -5,新的队头元素是 5
清空队列后,q.front = 2228,q.rear = 2228,q.front->next = 0
销毁队列后,q.front = 0,q.rear = 0
```

由 c3-2.h 和 c2-2.h 对比可见，单链队列和单链表的结构有相同之处。单链队列也是带有头结点的单链表，它的队头指针相当于单链表的头指针。因为队列操作是线性表操作的子集，所以 bo3-2.cpp 中的基本操作也可以用单链表的基本操作来代替。这样既可以充分利用现有资源，减小编程工作量，又可以更清楚地看出队列和线性表的内在联系和共性。bo3-3.cpp 是利用单链表的基本操作实现单链队列基本操作的程序。

单链队列和单链表的结构并不完全相同，因此只能是在单链队列的基本操作中调用单链表的基本操作。

```
// bo3-3.cpp 用单链表的基本操作实现链队列(存储结构由 c3-2.h 定义)的基本操作(9个)
typedef QElemType ElemType; // 定义单链表的元素类型为队列的元素类型
#define LinkList QueuePtr // 定义单链表的类型为相应的链队列的类型
#define LNode QNode
#include"bo2-2.cpp" // 单链表的基本操作
void InitQueue(LinkQueue &Q)
{ // 构造一个空队列 Q
  InitList(Q.front); // 以 Q.front 为头指针,构造空链表(调用单链表的基本操作)
  Q.rear = Q.front; // Q.rear 与 Q.front 共同指向链队列的头结点
}
void DestroyQueue(LinkQueue &Q)
{ // 销毁队列 Q(无论空否均可)
  DestroyList(Q.front); // 销毁 Q.front 为头指针的链表,且置 Q.front 为空
  Q.rear = Q.front; // 置 Q.rear 也为空
}
void ClearQueue(LinkQueue &Q)
{ // 将队列 Q 清为空队列
  ClearList(Q.front); // 清空以 Q.front 为头指针的链表,头结点的指针域为空
  Q.rear = Q.front; // Q.rear 也指向空队列的头结点
}
```

```
Status QueueEmpty(LinkQueue Q)
{ // 若队列 Q 为空队列,则返回 TRUE;否则返回 FALSE
  return ListEmpty(Q.front); // 以 Q.front 为头指针的单链表为空,则队列 Q 为空,反之亦然
}
int QueueLength(LinkQueue Q)
{ // 求队列 Q 的长度
  return ListLength(Q.front); // 队列 Q 的长度即为以 Q.front 为头指针的单链表的长度
}
Status GetHead(LinkQueue Q,QElemType &e)
{ // 若队列 Q 不空,则用 e 返回 Q 的队头元素,并返回 OK;否则返回 ERROR
  return GetElem(Q.front,1,e); // 队头元素即为以 Q.front 为头指针的单链表的第 1 个元素
}
void EnQueue(LinkQueue &Q,QElemType e)
{ // 插入元素 e 为队列 Q 的新的队尾元素
  ListInsert(Q.front,ListLength(Q.front)+1,e); // 在最后一个元素之后插入 e
}
Status DeQueue(LinkQueue &Q,QElemType &e)
{ // 若队列 Q 不空,删除 Q 的队头元素,用 e 返回其值,并返回 OK;否则返回 ERROR
  if(Q.front->next==Q.rear) // 队列仅有 1 个元素(删除的也是队尾元素)
    Q.rear=Q.front; // 令队尾指针指向头结点
  return ListDelete(Q.front,1,e); // 删除以 Q.front 为头指针的单链表的第 1 个元素
}
void QueueTraverse(LinkQueue Q,void(*visit)(QElemType))
{ // 从队头到队尾依次对队列 Q 中每个元素调用函数 visit()
  ListTraverse(Q.front,visit); // 依次对以 Q.front 为头指针的单链表的元素调用 visit()
}
```

main3-3.cpp 是检验 bo3-3.cpp 的程序。它的主函数及运行结果和 main3-2.cpp 的完全相同。bo3-3.cpp 和 bo3-2.cpp 相比,函数编写简单、思路清晰,但由于没有利用 Q.rear,EnQueue()函数的执行效率不高。

```
// main3-3.cpp 检验 bo3-3.cpp 的主程序
#include"c1.h"
typedef int QElemType; // 定义队列元素为整型
#include"c3-2.h" // 队列的链式存储结构
#include"bo3-3.cpp" // 用单链表的基本操作实现链队列的基本操作(9 个)
#include"func2-2.cpp" // 包括 equal()、comp()、print()、print1()和 print2()函数
#include"func3-2.cpp" // 主函数
```

**程序运行结果同 main3-2.cpp 的运行结果。**

### 3.4.2 循环队列——队列的顺序表示和实现

c3-3.h 采用循环队列:当队尾元素占据了存储空间的最后一个单元时,如再有新的元素入队,不是申请新的存储空间,而是将新元素插到存储空间的第 1 个单元,只要这个单元

为空(元素已出队)即可。通过头尾指针对存储空间 MAX_QSIZE 求余做到这一点,形成循环队列。

在循环队列中,队尾指针可能小于队头指针。入队元素时,队尾指针加 1。当队列满时,队尾指针等于队头指针,和队列空的条件一样。为了区别队满和队空,在循环队列中,少用一个存储单元。也就是在存储空间为 MAX_QSIZE 的循环队列中,最多只能存放 MAX_QSIZE－1 个元素。这样,队列空的条件仍为队尾指针等于队头指针,队列满的条件改为(队尾指针＋1)对 MAX_QSIZE 求余等于队头指针。

```
// c3-3.h 队列的顺序存储结构(循环队列)。在教科书第 64 页(见图 3-18)
#define MAX_QSIZE 5 // 最大队列长度 + 1
struct SqQueue
{ QElemType * base; // 初始化的动态分配存储空间
  int front; // 头指针,若队列不空,指向队列头元素
  int rear; // 尾指针,若队列不空,指向队列尾元素的下一个位置
};
```



图 3-18　循环队列的存储结构

```
// bo3-4.cpp 循环队列(存储结构由 c3-3.h 定义)的基本操作(9 个)
void InitQueue(SqQueue &Q)
{ // 构造一个空队列 Q。在教科书第 64 页(见图 3-19)
  Q.base = (QElemType * )malloc(MAX_QSIZE * sizeof(QElemType));
  if(!Q.base) // 存储分配失败
    exit(OVERFLOW);
  Q.front = Q.rear = 0;
}
```



图 3-19　空队列 Q

```
void DestroyQueue(SqQueue &Q)
{ // 销毁队列 Q,Q 不再存在(见图 3-20)
  if(Q.base) // 队列 Q 存在
    free(Q.base); // 释放 Q.base 所指的存储空间
  Q.base = NULL; // Q.base 不指向任何存储单元
  Q.front = Q.rear = 0;
}
```



图 3-20　销毁队列 Q

```
void ClearQueue(SqQueue &Q)
{ // 将队列 Q 清为空队列(见图 3-19)
  Q.front = Q.rear = 0;
}
Status QueueEmpty(SqQueue Q)
{ // 若队列 Q 为空队列,则返回 TRUE;否则返回 FALSE
  if(Q.front == Q.rear) // 队列空的标志
    return TRUE;
  else
    return FALSE;
}
Status GetHead(SqQueue Q,QElemType &e)
```

```
{ // 若队列 Q 不空,则用 e 返回 Q 的队头元素,并返回 OK;否则返回 ERROR
  if(Q.front == Q.rear) // 队列空
    return ERROR;
  e = Q.base[Q.front]; // 将队头元素的值赋给 e
  return OK;
}
Status EnQueue(SqQueue &Q,QElemType e)
{ // 插入元素 e 为队列 Q 的新的队尾元素。在教科书第 65 页(见图 3-21)
  if((Q.rear + 1) % MAX_QSIZE == Q.front) // 队列满
    return ERROR;
  Q.base[Q.rear] = e; // 将 e 插在队尾
  Q.rear = (Q.rear + 1) % MAX_QSIZE;
  // 队尾指针 + 1 后对 MAX_QSIZE 取余
  return OK;
}
int QueueLength(SqQueue Q)
{ // 返回队列 Q 的元素个数,即队列的长度。在教科书第 64 页
  return(Q.rear - Q.front + MAX_QSIZE) % MAX_QSIZE;
}
Status DeQueue(SqQueue &Q,QElemType &e) // 在教科书第 65 页
{ // 若队列 Q 不空,则删除 Q 的队头元素,用 e 返回其值,
  // 并返回 OK;否则返回 ERROR(见图 3-22)
  if(Q.front == Q.rear) // 队列空
    return ERROR;
  e = Q.base[Q.front]; // 将队头元素的值赋给 e
  Q.front = (Q.front + 1) % MAX_QSIZE; // 移动队头指针
  return OK;
}
void QueueTraverse(SqQueue Q,void( * visit)(QElemType))
{ // 从队头到队尾依次对队列 Q 中每个元素调用函数 visit()
  int i = Q.front; // i 最初指向队头元素
  while(i! = Q.rear) // i 指向队列 Q 中的元素
  { visit(Q.base[i]); // 对 i 所指元素调用函数 visit()
    i = (i + 1) % MAX_QSIZE; // i 指向下一个元素
  }
  printf("\n");
}
```


(a) 调用EnQueue()前


(b) 调用EnQueue()后

图 3-21 调用 EnQueue()示例


(a) 调用DeQueue()前


(b) 调用DeQueue()后

图 3-22 调用 DeQueue()示例

```
// main3-4.cpp 循环队列 检验 bo3-4.cpp 的主程序
#include"c1.h"
typedef int QElemType; // 定义队列元素为整型
#include"c3-3.h" // 队列的顺序存储结构(循环队列)
#include"bo3-4.cpp" // 循环队列存储结构的基本操作(9 个)
#define ElemType QElemType // 将 func2-2.cpp 中的 ElemType 类型定义为 QElemType 类型
```

```
#include"func2-2.cpp" // 包括 equal()、comp()、print()、print1()和 print2()函数
void main()
{
  Status j;
  int i = 0,m;
  QElemType d;
  SqQueue Q;
  InitQueue(Q); // 初始化队列 Q,失败则退出
  printf("初始化队列后,队列空否? %u(1:空 0:否)\n",QueueEmpty(Q));
  printf("请输入整型队列元素(不超过 %d 个), - 1 为提前结束符: ",MAX_QSIZE - 1);
  do
  { scanf("%d",&d); // 由键盘输入整型队列元素
    if(d == - 1) // 输入的是提前结束符
      break; // 退出输入数据循环
    i ++; // 计数器 + 1
    EnQueue(Q,d); // 入队输入的元素
  }while(i<MAX_QSIZE - 1); // 队列元素的个数不超过允许的范围
  printf("队列长度为 %d,",QueueLength(Q));
  printf("现在队列空否? %u(1:空 0:否)\n",QueueEmpty(Q));
  printf("连续 %d 次由队头删除元素,队尾插入元素: \n",MAX_QSIZE);
  for(m = 1;m<= MAX_QSIZE;m ++ )
  { DeQueue(Q,d); // 删除队头元素,其值赋给 d
    printf("删除的元素是 %d,请输入待插入的元素: ",d);
    scanf("%d",&d); // 输入要入队的元素给 d
    EnQueue(Q,d); // 将 d 入队
  }
  m = QueueLength(Q); // m 为队列 Q 的长度
  printf("现在队列中的元素为");
  QueueTraverse(Q,print); // 从队头到队尾依次对队列 Q 的每个元素调用函数 print()
  printf("共向队尾插入了 %d 个元素。",i + MAX_QSIZE);
  if(m - 2>0)
    printf("现在由队头删除 %d 个元素,",m - 2);
  while(QueueLength(Q)>2)
  { DeQueue(Q,d); // 删除队头元素,其值赋给 d
    printf("删除的元素值为 %d,",d);
  }
  j = GetHead(Q,d); // 将队头元素赋给 d
  if(j) // 队列 Q 不空
    printf("现在队头元素为 %d\n",d);
  ClearQueue(Q); // 清空队列 Q
  printf("清空队列后,队列空否? %u(1:空 0:否)\n",QueueEmpty(Q));
  DestroyQueue(Q); // 销毁队列 Q
}
```

**程序运行结果：**

```
初始化队列后，队列空否？1(1:空 0:否)
请输入整型队列元素(不超过4个)，-1为提前结束符：1 2 3 -1↙
队列长度为3，现在队列空否？0(1:空 0:否)
连续5次由队头删除元素，队尾插入元素：
删除的元素是1，请输入待插入的元素：4↙
删除的元素是2，请输入待插入的元素：5↙
删除的元素是3，请输入待插入的元素：6↙
删除的元素是4，请输入待插入的元素：7↙
删除的元素是5，请输入待插入的元素：8↙
现在队列中的元素为6 7 8
共向队尾插入了8个元素。现在由队头删除1个元素，删除的元素值为6，现在队头元素为7
清空队列后，队列空否？1(1:空 0:否)
```

由于队列要在表的一端插入元素，在表的另一端删除元素，故采用链式存储结构比较好。可以从根本上免除移动队列元素的操作，且节约存储空间。

# 第 4 章

# 串

## 4.1 串类型的定义

在 C 语言中，字符串存于字符型数组中。无论数组有多大，都用数值 0 表示串结束。图 4-1 说明了字符串"but"在 C 语言中的存储结构。其中数组 a 的定义为：

```
char a[10];
```

| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] |
|---|---|---|---|---|---|---|---|---|---|
| b | u | t | \0 | | | | | | |

有效字符（b u t）　a

图 4-1 "but"在 C 语言中的存储结构

C 语言还在库函数 string.h 中提供了许多串处理的基本操作，如求串长函数 strlen()、串复制函数 strcpy()等。

算法语言本身提供的字符串存储结构及其基本操作不一定能满足实际应用的需要，往往还要根据具体情况另外定义字符串的存储结构及基于该存储结构的基本操作。

和算法 2.1 类似，func4-1.cpp 中的算法 4.1 的形参 S 和 T 的类型是 String(串)，String 也是抽象的串类型。算法 4.1 中所涉及的函数都是串的基本操作，如 InitString()、StrLength()等，不涉及具体的数据存储结构。func4-1.cpp 中的另一个基本操作算法，Replace()函数也与数据的存储结构无关。这样，它们就可以应用到任何一种具体的串存储结构中。main4-1.cpp 和 main4-2.cpp 都调用了 func4-1.cpp。

```
// func4-1.cpp 与存储结构无关的两个基本操作
int Index(String S,String T,int pos) // 算法 4.1
{ // T 为非空串。若主串 S 中第 pos 个字符之后存在与 T 相等的子串，
  // 则返回第一个这样的子串在 S 中的位置；否则返回 0
  int n,m,i;
  String sub;
  InitString(sub); // 新增
  if(pos>0)
  { n=StrLength(S); // 主串 S 的长度
    m=StrLength(T); // 模式串 T 的长度
    i=pos;
    while(i<=n-m+1) // i 从串 S 的 pos 到倒数第 m 个
    { SubString(sub,S,i,m); // 子串 sub 是从主串 S 的第 i 个字符起，长度为 m 的子串
```

```
        if(StrCompare(sub,T)! = 0) // 子串 sub 不等于模式串 T
          ++ i; // 继续向后比较
        else // 子串 sub 等于模式串 T
          return i; // 返回模式串 T 的第 1 个字符在主串 S 中的位置
      }
    }
    return 0; // 主串 S 中不存在与模式 T 相等的子串
  }
  Status Replace(String &S,String T,String V)
  { // 初始条件:串 S、T 和 V 存在,串 T 是非空串
    // 操作结果:用串 V 替换主串 S 中出现的所有与串 T 相等的不重叠的子串
    int i = 1; // 从串 S 的第一个字符起查找串 T
    Status k;
    if(StrEmpty(T)) // T 是空串
      return ERROR;
    while(i)
    { i = Index(S,T,i); // 结果 i 为从上一个 i 之后找到的子串 T 的位置
      if(i) // 串 S 中存在串 T
      { StrDelete(S,i,StrLength(T)); // 删除串 T
        k = StrInsert(S,i,V); // 在原串 T 的位置插入串 V
        if(!k) // 不能完全插入(定长顺序存储结构有可能发生这种情况)
          return ERROR;
        i += StrLength(V); // 在插入的串 V 后面继续查找串 T
      }
    };
    return OK;
  }
```

# 4.2 串的表示和实现

## 4.2.1 定长顺序存储表示

**// c4-1.h 串的定长顺序存储结构。在教科书第 73 页**(见图 4-2)

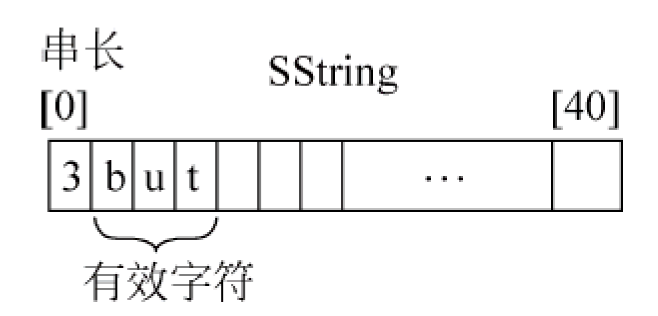



图 4-2 定长顺序存储结构

```
#define MAX_STR_LEN 40 // 用户可在 255(1 个字节所能表示的最大整数)以内定义最大串长
typedef unsigned char SString[MAX_STR_LEN + 1]; // 0 号单元存放串的长度
```

```
// bo4-1.cpp 串采用定长顺序存储结构(由 c4-1.h 定义)的基本操作(12 个),
// 包括算法 4.2、算法 4.3 和算法 4.5。SString 是数组,故不需要引用类型
```

```
#define DestroyString ClearString // DestroyString()与 ClearString()作用相同
#define InitString ClearString // InitString()与 ClearString()作用相同
Status StrAssign(SString T,char* chars)
{ // 生成一个其值等于 chars 的串 T
  int i;
  if(strlen(chars)>MAX_STR_LEN) // chars 的长度大于最大串长
    return ERROR;
  else
  { T[0] = strlen(chars); // 0 号单元存放串的长度
    for(i = 1;i<= T[0];i++) // 从 1 号单元起复制串的内容
      T[i] = *(chars + i - 1);
    return OK;
  }
}
void StrCopy(SString T,SString S)
{ // 由串 S 复制得串 T
  int i;
  for(i = 0;i<= S[0];i++)
    T[i] = S[i];
}
Status StrEmpty(SString S)
{ // 若 S 为空串,则返回 TRUE;否则返回 FALSE
  if(S[0] == 0)
    return TRUE;
  else
    return FALSE;
}
int StrCompare(SString S,SString T)
{ // 初始条件:串 S 和串 T 存在
  // 操作结果:若 S>T,则返回值>0;若 S = T,则返回值 = 0;若 S<T,则返回值<0
  int i;
  for(i = 1;i<= S[0]&&i<= T[0];++i)
    if(S[i]!= T[i])
      return S[i] - T[i];
  return S[0] - T[0];
}
int StrLength(SString S)
{ // 返回串 S 的元素个数
  return S[0];
}
void ClearString(SString S)
{ // 初始条件:串 S 存在。操作结果:将 S 清为空串(见图 4-3)
  S[0] = 0; // 令串长为零
}
Status Concat(SString T,SString S1,SString S2) // 算法 4.2 修改
```

| 0 | … | |
|---|---|---|
| [0] | | [40] |

S

图 4-3　空串 S

```
{ // 用 T 返回 S1 和 S2 连接而成的新串。若未截断,则返回 TRUE;否则返回 FALSE
  int i;
  if(S1[0] + S2[0]<= MAX_STR_LEN) // 未截断
  { for(i = 1;i<= S1[0];i ++ )
      T[i] = S1[i];
    for(i = 1;i<= S2[0];i ++ )
      T[S1[0] + i] = S2[i];
    T[0] = S1[0] + S2[0];
    return TRUE;
  }
  else // 截断 S2
  { for(i = 1;i<= S1[0];i ++ )
      T[i] = S1[i];
    for(i = 1;i<= MAX_STR_LEN - S1[0];i ++ ) // 到串长为止
      T[S1[0] + i] = S2[i];
    T[0] = MAX_STR_LEN;
    return FALSE;
  }
}
Status SubString(SString Sub,SString S,int pos,int len)
{ // 用 Sub 返回串 S 的自第 pos 个字符起长度为 len 的子串。算法 4.3
  int i;
  if(pos<1||pos>S[0]||len<0||len>S[0] - pos + 1) // pos 和 len 的值超出范围
    return ERROR;
  for(i = 1;i<= len;i ++ )
    Sub[i] = S[pos + i - 1];
  Sub[0] = len;
  return OK;
}
int Index1(SString S,SString T,int pos)
{ // 返回子串 T 在主串 S 中第 pos 个字符之后的位置。若不存在,则函数值为 0。
  // 其中,T 非空,1≤pos≤StrLength(S)。算法 4.5
  int i,j; // 指示主串 S 和子串 T 的当前比较字符
  if(1<= pos&&pos<= S[0]) // pos 的范围合适
  { i = pos; // 从主串 S 的第 pos 个字符开始和子串 T 的第 1 个字符比较
    j = 1;
    while(i<= S[0]&&j<= T[0])
      if(S[i] == T[j]) // 当前两字符相等
      {  ++ i; // 继续比较后继字符
         ++ j;
      }
      else // 当前两字符不相等
      { i = i - j + 2; // 两指针后退重新开始匹配
        j = 1;
      }
```

```
    if(j>T[0]) // 主串 S 中存在子串 T
      return i - T[0];
    else // 主串 S 中不存在子串 T
      return 0;
  }
  else // pos 的范围不合适
    return 0;
}
Status StrInsert(SString S,int pos,SString T)
{ // 初始条件：串 S 和 T 存在,1≤pos≤StrLength(S) + 1
  // 操作结果：在串 S 的第 pos 个字符之前插入串 T。完全插入返回 TRUE,部分插入返回 FALSE
  int i;
  if(pos<1||pos>S[0] + 1) // pos 超出范围
    return ERROR;
  if(S[0] + T[0]<= MAX_STR_LEN) // 完全插入
  { for(i = S[0];i>= pos;i -- ) // 移动串 S 中位于 pos 之后的字符
      S[i + T[0]] = S[i]; // 串 S 向后移串 T 的长度,为插入串 T 准备空间
    for(i = pos;i<pos + T[0];i ++ ) // 在串 S 中插入串 T
      S[i] = T[i - pos + 1];
    S[0] += T[0]; // 更新串 S 的长度
    return TRUE; // 完全插入的标记
  }
  else // 部分插入
  { for(i = MAX_STR_LEN;i>= pos + T[0];i -- ) // 移动串 S 中位于 pos 之后且移后仍在串内的字符
      S[i] = S[i - T[0]];
    for(i = pos;i<pos + T[0]&&i<= MAX_STR_LEN;i ++ ) // 在串 S 中插入串 T(也可能是部分插入)
      S[i] = T[i - pos + 1];
    S[0] = MAX_STR_LEN; // 串 S 的长度为串的最大长度
    return FALSE; // 部分插入的标记
  }
}
Status StrDelete(SString S,int pos,int len)
{ // 初始条件：串 S 存在,1≤pos≤StrLength(S) - len + 1
  // 操作结果：从串 S 中删除自第 pos 个字符起长度为 len 的子串
  int i;
  if(pos<1||pos>S[0] - len + 1||len<0) // pos 和 len 的值超出范围
    return ERROR; // 删除不成功的标记
  for(i = pos + len;i<= S[0];i ++ ) // 对于删除的子串之后的所有字符
    S[i - len] = S[i]; // 向前移动删除子串的长度
  S[0] -= len; // 更新串 S 的长度
  return OK; // 删除成功的标记
}
void StrPrint(SString S)
{ // 输出字符串 S。新增
  int i;
```

```
  for(i=1;i<=S[0];i++)
    printf("%c",S[i]);
  printf("\n");
}

// main4-1.cpp 检验 bo4-1.cpp 的主程序
#include"c1.h"
#include"c4-1.h" // 串的定长顺序存储结构
#include"bo4-1.cpp" // 定长顺序存储结构的基本操作(12 个)
typedef SString String; // 定义抽象数据类型 String 为 SString 类型
#include"func4-1.cpp" // 与存储结构无关的两个基本操作
void main()
{
  int i,j;
  Status k;
  char s,c[MAX_STR_LEN+1]; // c 中包括串结束符
  SString t,s1,s2;
  printf("请输入串 s1: ");
  gets(c); // 由键盘输入字符串给 c
  k=StrAssign(s1,c); // 将字符串 c 转为 SString 类型,存入 s1
  if(!k) // 本例由于 c 的长度所限,串长超过 MAX_STR_LEN 的现象不会发生
  { printf("串长超过 MAX_STR_LEN(=%d)\n",MAX_STR_LEN);
    exit(OVERFLOW);
  }
  printf("串长为%d,串空否?%d(1:是 0:否)\n",StrLength(s1),StrEmpty(s1));
  StrCopy(s2,s1); // 复制串 s1 生成串 s2
  printf("复制 s1 生成的串为");
  StrPrint(s2); // 输出串 s2
  printf("请输入串 s2: ");
  gets(c); // 由键盘输入字符串给 c
  StrAssign(s2,c); // 将字符串 c 转为 SString 类型,存入 s1。可不要返回值
  i=StrCompare(s1,s2); // 比较串 s1 和串 s2
  if(i<0)
    s='<';
  else if(i==0)
    s='=';
  else
    s='>';
  printf("串 s1 %c 串 s2\n",s);
  k=Concat(t,s1,s2); // 由串 s1 连接串 s2 生成串 t
  printf("串 s1 连接串 s2 得到的串 t 为");
  StrPrint(t); // 输出串 t
  if(k==FALSE)
    printf("串 t 有截断\n");
```

```
  ClearString(s1); // 清空串 s1
  printf("清为空串后,串 s1 为");
  StrPrint(s1); // 输出串 s1
  printf("串长为 %d,串空否?%d(1:是 0:否)\n",StrLength(s1),StrEmpty(s1));
  printf("求串 t 的子串,请输入子串的起始位置,子串长度:");
  scanf("%d,%d",&i,&j);
  k = SubString(s2,t,i,j); // 串 s2 为串 t 的第 i 个字符起,长度为 j 的子串
  if(k) // 串 s2 存在
  { printf("子串 s2 为");
    StrPrint(s2); // 输出串 s2
  }
  printf("从串 t 的第 pos 个字符起,删除 len 个字符,请输入 pos,len:");
  scanf("%d,%d",&i,&j);
  StrDelete(t,i,j); // 将串 t 的第 i 个字符起的 j 个字符删除
  printf("删除后的串 t 为");
  StrPrint(t); // 输出串 t
  i = StrLength(s2)/2; // i 为串 s2 长度的一半取整
  StrInsert(s2,i,t); // 在串 s2 的第 i 个字符之前插入串 t
  printf("在串 s2 的第 %d 个字符之前插入串 t 后,串 s2 为",i);
  StrPrint(s2); // 输出串 s2
  i = Index1(s2,t,1); // 从串 s2 的第 1 个字符起查找串 t
  printf("s2 的第 %d 个字符起和 t 第一次匹配\n",i);
  i = Index(s2,t,1); // 从串 s2 的第 1 个字符起查找串 t(另一种方法)
  printf("s2 的第 %d 个字符起和 t 第一次匹配\n",i);
  SubString(t,s2,1,1); // 串 t 为串 s2 的第 1 个字符
  printf("串 t 为");
  StrPrint(t); // 输出串 t
  Concat(s1,t,t); // 串 s1 为 2 个串 t
  printf("串 s1 为");
  StrPrint(s1); // 输出串 s1
  k = Replace(s2,t,s1); // 将串 s2 中的所有不重叠的串 t,用串 s1 替换
  if(k) // 替换成功
  { printf("用串 s1 取代串 s2 中和串 t 相同的不重叠的串后,串 s2 为");
    StrPrint(s2); // 输出串 s2
  }
  DestroyString(s2); // 销毁操作与清空操作作用相同
}
```

**程序运行结果:**

```
请输入串 s1:ABCD↙
串长为 4,串空否?0(1:是 0:否)
复制 s1 生成的串为 ABCD
请输入串 s2:123456↙
串 s1>串 s2
```

```
串 s1 连接串 s2 得到的串 t 为 ABCD123456
清为空串后,串 s1 为
串长为 0,串空否? 1(1:是 0:否)
求串 t 的子串,请输入子串的起始位置,子串长度: 3,7↙
子串 s2 为 CD12345
从串 t 的第 pos 个字符起,删除 len 个字符,请输入 pos,len: 4,4↙
删除后的串 t 为 ABC456
在串 s2 的第 3 个字符之前插入串 t 后,串 s2 为 CDABC45612345
s2 的第 3 个字符起和 t 第一次匹配
s2 的第 3 个字符起和 t 第一次匹配
串 t 为 C
串 s1 为 CC
用串 s1 取代串 s2 中和串 t 相同的不重叠的串后,串 s2 为 CCDABCC45612345
```

## 4.2.2 堆分配存储表示

```
// c4-2.h 串的堆分配存储结构。在教科书第 75 页(见图 4-4)
struct HString
{ char *ch; // 若是非空串,则按串长分配存储区;否则 ch 为 NULL
  int length; // 串长度
};
```



图 4-4 堆分配存储结构

```
// bo4-2.cpp 串采用堆分配存储结构(由 c4-2.h 定义)的基本操作(12 个)。包括算法 4.4
#define DestroyString ClearString // DestroyString()与 ClearString()作用相同
void InitString(HString &S)
{ // 初始化(产生空串)字符串 S。新增(见图 4-5)
  S.length = 0;
  S.ch = NULL;
}
void ClearString(HString &S)
{ // 将 S 清为空串。在教科书第 77 页(见图 4-5)
  free(S.ch); // 释放 S.ch 所指空间
  InitString(S); // 初始化串 S
}
void StrAssign(HString &T,char *chars)
{ // 生成一个其值等于串常量 chars 的串 T。在教科书第 76 页(见图 4-6)
  int i,j;
  if(T.ch) // T 指向某存储单元
    free(T.ch); // 释放 T 原有存储空间
  i = strlen(chars); // 求 chars 的长度 i
  if(!i) // chars 的长度为 0
    InitString(T); // 生成空串
  else // chars 的长度不为 0
```



图 4-5 空串 S



图 4-6 "but"的堆存储结构

```
  { T.ch = (char * )malloc(i * sizeof(char)); // 分配串存储空间
    if(!T.ch) // 分配串存储空间失败
      exit(OVERFLOW);
    for(j = 0;j<i;j ++ ) // 分配串存储空间成功后，复制串 chars[]到串 T
      T.ch[j] = chars[j];
    T.length = i; // 更新串 T 的长度
  }
}
void StrCopy(HString &T,HString S)
{ // 初始条件：串 S 存在。操作结果：由串 S 复制得串 T
  int i;
  if(T.ch) // 串 T 不空
    free(T.ch); // 释放串 T 原有存储空间
  T.ch = (char * )malloc(S.length * sizeof(char)); // 分配串存储空间
  if(!T.ch) // 分配串存储空间失败
    exit(OVERFLOW);
  for(i = 0;i<S.length;i ++ ) // 从第 1 个字符到最后一个字符
    T.ch[i] = S.ch[i]; // 逐一复制字符
  T.length = S.length; // 复制串长
}
Status StrEmpty(HString S)
{ // 初始条件：串 S 存在。操作结果：若串 S 为空，则返回 TRUE；否则返回 FALSE
  if(S.length == 0&&S.ch == NULL) // 空串标志
    return TRUE;
  else
    return FALSE;
}
int StrCompare(HString S,HString T)
{ // 若串 S>串 T，则返回值>0；若 S = T，则返回值 = 0；若 S<T，则返回值<0。在教科书第 77 页
  int i;
  for(i = 0;i<S.length&&i<T.length;++ i) // 在有效范围内
    if(S.ch[i]! = T.ch[i]) // 逐一比较字符
      return S.ch[i] - T.ch[i]; // 不相等，则返回 2 字符 ASCII 码之差
  return S.length - T.length; // 在有效范围内，字符相等，但长度不等，返回长度之差
}
int StrLength(HString S)
{ // 返回串 S 的元素个数，称为串的长度。在教科书第 77 页
  return S.length;
}
void Concat(HString &T,HString S1,HString S2)
{ // 用串 T 返回由串 S1 和串 S2 连接而成的新串。在教科书第 77 页
  int i;
  if(T.ch) // 串 T 不空
    free(T.ch); // 释放串 T 原有存储空间
  T.length = S1.length + S2.length; // 串 T 的长度 = 串 S1 的长度 + 串 S2 的长度
```

```
    T.ch = (char * )malloc(T.length * sizeof(char)); // 分配串 T 的存储空间
    if(!T.ch) // 分配串存储空间失败
      exit(OVERFLOW);
    for(i = 0;i<S1.length;i ++ ) // 将串 S1 的字符逐一复制给串 T
      T.ch[i] = S1.ch[i];
    for(i = 0;i<S2.length;i ++ ) // 将串 S2 的字符逐一复制给串 T(接在串 S1 的字符之后)
      T.ch[S1.length + i] = S2.ch[i];
}
Status SubString(HString &Sub,HString S,int pos,int len)
{ // 用 Sub 返回串 S 的第 pos 个字符起长度为 len 的子串。
  // 其中,1≤pos≤StrLength(S)且 0≤len≤StrLength(S) - pos + 1。在教科书第 77 页
  int i;
  if(pos<1||pos>S.length||len<0||len>S.length - pos + 1) // pos 和 len 的值超出范围
    return ERROR; // 无法用 Sub 返回子串
  if(Sub.ch) // 串 Sub 不空
    free(Sub.ch); // 释放串 Sub 原有存储空间
  if(!len) // 空子串
    InitString(Sub); // 初始化串 Sub
  else // 完整子串
  { Sub.ch = (char * )malloc(len * sizeof(char)); // 分配串 Sub 的存储空间
    if(!Sub.ch) // 分配串存储空间失败
      exit(OVERFLOW);
    for(i = 0;i<= len - 1;i ++ ) // 将串 S 第 pos 个字符起长度为 len 的子串的字符逐一复制给串 Sub
      Sub.ch[i] = S.ch[pos - 1 + i];
    Sub.length = len; // 串 Sub 的长度
  }
  return OK;
}
Status StrInsert(HString &S,int pos,HString T) // 算法 4.4
{ // 1≤pos≤StrLength(S) + 1。在串 S 的第 pos 个字符之前插入串 T
  int i;
  if(pos<1||pos>S.length + 1) // pos 不合法
    return ERROR;
  if(T.length) // T 非空
  { S.ch = (char * )realloc(S.ch,(S.length + T.length) * sizeof(char)); // 重分 S 存储空间
    if(!S.ch) // 重新分配串 S 的存储空间失败
      exit(OVERFLOW);
    for(i = S.length - 1;i>= pos - 1;-- i) // 为插入 T 而腾出位置
      S.ch[i + T.length] = S.ch[i];
    for(i = 0;i<T.length;i ++ ) // 插入 T
      S.ch[pos - 1 + i] = T.ch[i];
    S.length += T.length; // 更新串 S 的长度
  }
  return OK;
}
```

```
Status StrDelete(HString &S,int pos,int len)
{ // 从串 S 中删除第 pos 个字符起长度为 len 的子串
  int i;
  if(S.length<pos+len-1) // pos 和 len 的值超出范围
    return ERROR;
  for(i=pos-1;i<=S.length-len;i++) // 将待删除子串之后的字符逐一前移
    S.ch[i]=S.ch[i+len];
  S.length-=len; // 更新串 S 的长度
  S.ch=(char*)realloc(S.ch,S.length*sizeof(char)); // 重新分配串 S 的存储空间(减少)
  return OK;
}
void StrPrint(HString S)
{ // 输出字符串 S。新增
  int i;
  for(i=0;i<S.length;i++)
    printf("%c",S.ch[i]);
  printf("\n");
}
```

```
// main4-2.cpp 检验 bo4-2.cpp 的主程序
#include"c1.h"
#include"c4-2.h" // 串的堆分配存储结构
#include"bo4-2.cpp" // 串采用堆分配存储结构的基本操作(12 个)
typedef HString String; // 定义抽象数据类型 String 为 HString 类型
#include"func4-1.cpp" // 与存储结构无关的两个基本操作
void main()
{
  int i;
  char c,*p="God bye!",*q="God luck!";
  HString t,s,r;
  InitString(t); // HString 类型必须初始化
  InitString(s);
  InitString(r);
  StrAssign(t,p); // 将字符串 p 的内容转成 HString 类型,赋给 t
  printf("串 t 为");
  StrPrint(t); // 输出串 t
  printf("串长为%d,串空否?%d(1:空 0:否)\n",StrLength(t),StrEmpty(t));
  StrAssign(s,q); // 将字符串 q 的内容转成 HString 类型,赋给 s
  printf("串 s 为");
  StrPrint(s); // 输出串 s
  i=StrCompare(s,t); // 比较串 s 和串 t 的大小
  if(i<0)
    c='<';
  else if(i==0)
    c='=';
```

```
    else
      c = '>';
    printf("串 s%c 串 t\n",c);
    Concat(r,t,s); // 连接串 t 和串 s,得到串 r
    printf("串 t 连接串 s 产生的串 r 为");
    StrPrint(r); // 输出串 r
    StrAssign(s,"oo"); // 将字符串"oo"转成 HString 类型,赋给 s
    printf("串 s 为");
    StrPrint(s); // 输出串 s
    StrAssign(t,"o"); // 将字符串"o"转成 HString 类型,赋给 t
    printf("串 t 为");
    StrPrint(t); // 输出串 t
    Replace(r,t,s); // 将串 r 中和串 t 相同的子串用串 s 代替
    printf("把串 r 中和串 t 相同的子串用串 s 代替后,串 r 为");
    StrPrint(r); // 输出串 r
    ClearString(s); // 清空串 s
    printf("串 s 清空后,串长为%d,空否?%d(1:空 0:否)\n",StrLength(s),StrEmpty(s));
    SubString(s,r,6,4); // 生成的串 s 为从串 r 的第 6 个字符起的 4 个字符
    printf("串 s 为从串 r 的第 6 个字符起的 4 个字符,长度为%d,串 s 为",s.length);
    StrPrint(s); // 输出串 s
    StrCopy(t,r); // 由串 r 复制得串 t
    printf("由串 r 复制得串 t,串 t 为");
    StrPrint(t); // 输出串 t
    StrInsert(t,6,s); // 在串 t 的第 6 个字符前插入串 s
    printf("在串 t 的第 6 个字符前插入串 s 后,串 t 为");
    StrPrint(t); // 输出串 t
    StrDelete(t,1,5); // 从串 t 的第 1 个字符起删除 5 个字符
    printf("从串 t 的第 1 个字符起删除 5 个字符后,串 t 为");
    StrPrint(t); // 输出串 t
    printf("%d 是从串 t 的第 1 个字符起,和串 s 相同的第 1 个子串的位置\n",Index(t,s,1));
    printf("%d 是从串 t 的第 2 个字符起,和串 s 相同的第 1 个子串的位置\n",Index(t,s,2));
    DestroyString(t); // 销毁操作同清空
  }
```

**程序运行结果:**

```
串 t 为 God bye!
串长为 8,串空否? 0(1:空 0:否)
串 s 为 God luck!
串 s>串 t
串 t 连接串 s 产生的串 r 为 God bye!God luck!
串 s 为 oo
串 t 为 o
把串 r 中和串 t 相同的子串用串 s 代替后,串 r 为 Good bye!Good luck!
```

```
串 s 清空后,串长为 0,空否? 1(1:空 0:否)
串 s 为从串 r 的第 6 个字符起的 4 个字符,长度为 4,串 s 为 bye!
由串 r 复制得串 t,串 t 为 Good bye!Good luck!
在串 t 的第 6 个字符前插入串 s 后,串 t 为 Good bye!bye!Good luck!
从串 t 的第 1 个字符起删除 5 个字符后,串 t 为 bye!bye!Good luck!
1 是从串 t 的第 1 个字符起,和串 s 相同的第 1 个子串的位置
5 是从串 t 的第 2 个字符起,和串 s 相同的第 1 个子串的位置
```

串的堆分配存储结构(由 c4-2.h 定义)根据串的长度,动态地分配存储空间。这样既保证满足需要,又不浪费空间,对串长也没有限制。串的定长存储结构(由 c4-1.h 定义)就没有这样灵活了。在 Concat()、StrInsert()和 Replace()中,总要检查串是否被截断。且对于短串的情况,空间浪费较大。故堆分配存储结构较好。

## 4.3　串的模式匹配算法

### 4.3.1　求子串位置的定位函数 Index(S,T,pos)

串的模式匹配的一般方法(见图 4-7)如算法 4.5(在 bo4-1.cpp 中)所示:由主串 S 的第 pos 个字符起,检验是否存在子串 T。首先令 i 等于 pos(i 为 S 中当前待比较字符的位序),j 等于 1(j 为 T 中当前待比较字符的位序),如果 S 的第 i 个字符与 T 的第 j 个字符相同,则 i、j 各加 1 继续比较,直至 T 的最后一个字符(找到)。如果在比较期间出现了不同(没找到),则令 i 等于 pos+1,j 等于 1,由 pos 的下一个位置起,继续查找是否存在子串 T。

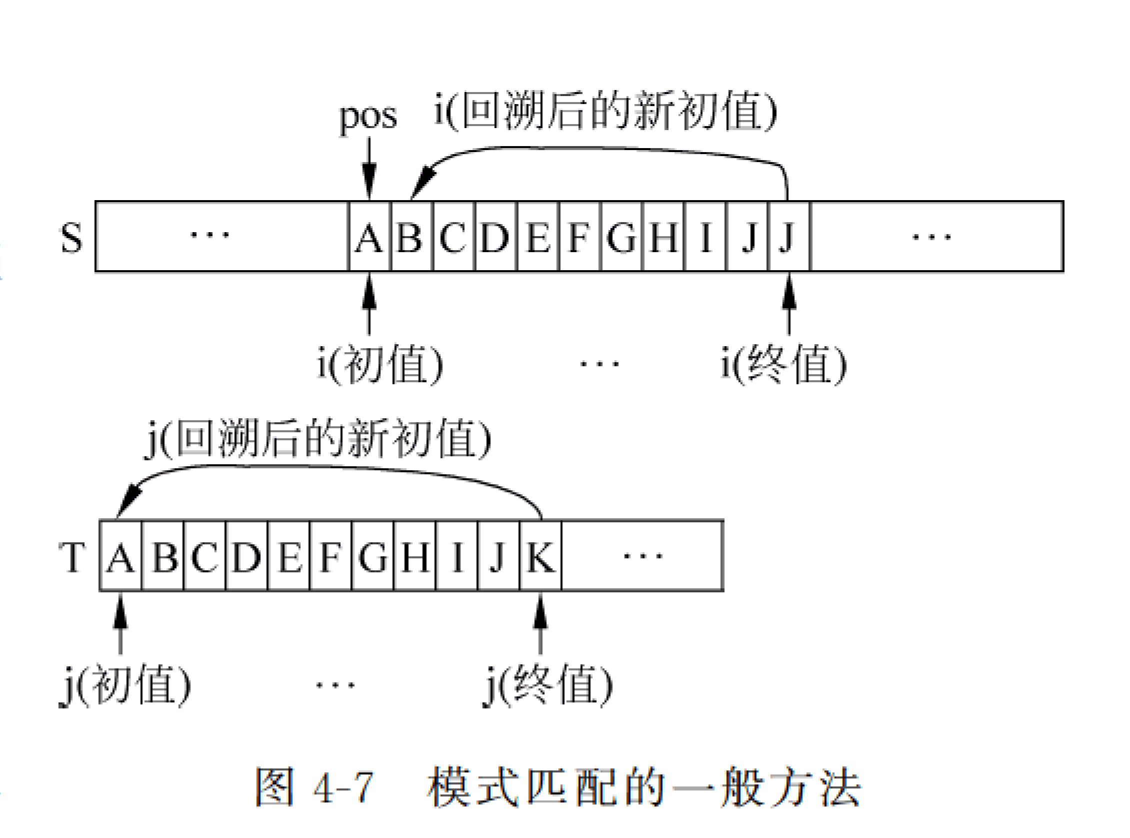



图 4-7　模式匹配的一般方法

### 4.3.2　模式匹配的一种改进算法

在算法 4.5 中,主串 S 的指针 i 总要回溯,特别是在如图 4-7 所示的有较多字符匹配而又不完全匹配的情况下,回溯得更多。这时,主串 S 的每个字符要进行多次比较,显然效率较低。

如果能使主串 S 的指针 i 不回溯,也就是使主串 S 的每个字符只进行一次比较,效率会大为提高。这是可以做到的。仍以图 4-7 为例,当检测到主串 S 中第 i(终值)个字符与模式串 T 中第 j(终值)个字符不匹配时,主串 S 中第 i(终值)−1,i(终值)−2,…,i(终值)−j(终值)个字符分别和子串 T 中第 j(终值)−1,j(终值)−2,…,1 个字符相等。也就是说,当 S 和 T 在第 i(终值)个字符处不匹配时,i(初值)之后到 i(终值)之前的字符中不再有 A,故 i 可仍保持在终值处不动,j 回溯到子串 T 的第 1 个字符处与 i 的当前字符继续进行比较。j 回溯到第几个字符是由子串 T 的模式决定的。算法 4.7 根据子串 T 生成的 next 数组指示

j 回溯到第几个字符。next 数组的意义是这样的：如果 next[j]=k，则当子串 T 的第 j 个字符与主串 S 的第 i 个字符“失配”时，S 的第 i 个字符继续与 T 的第 k 个字符进行比较即可。T 的第 k 个字符之前的那些字符均与 S 的第 i 个字符之前的字符匹配。以教科书中图 4.5 为例，设子串 T 为“abaabcac”。当 T 的第 5 个字符与 S 的第 i 个字符失配时，S 的第 i−1 个字符一定是 a，和 T 的第 4 个字符相等。它和 T 的第 1 个字符相等。这样，S 的第 i 个字符和 T 的第 2 个字符开始比较即可。所以，对于模式串“abaabcac”，next[5]=2，详见图 4-8。

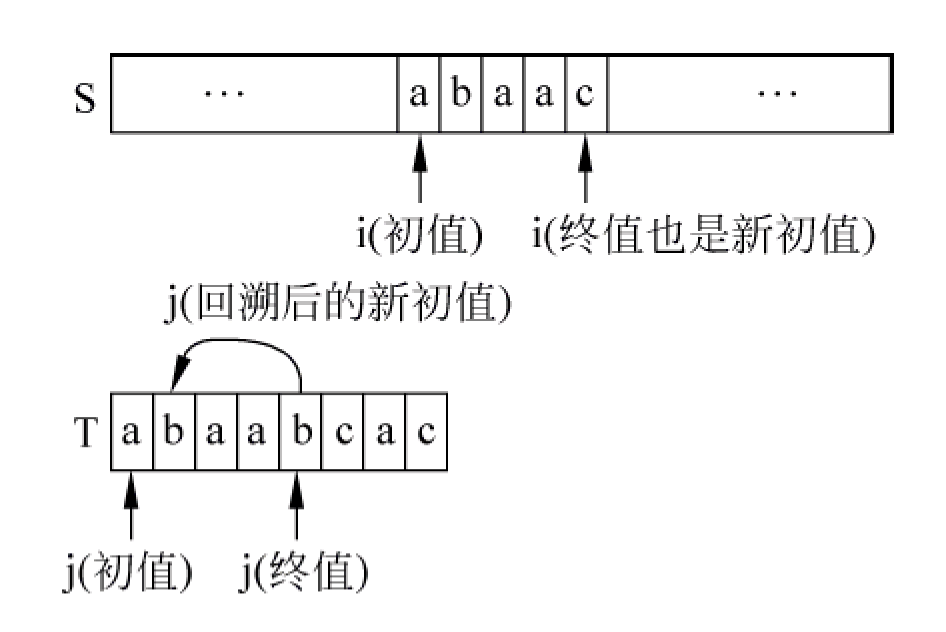



图 4-8　串“abaabcac”的数组 next[5]=2 的由来

算法 4.7 求子串的数组 next[]还有可改进之处。以图 4-8 为例：如果 T 的第 5 个字符与 S 的第 i 个字符失配，则 S 的第 i 个字符一定不是 b。这样，尽管 S 的第 i−1 个字符是 a，和 T 的第 1 个字符相等，但 S 的第 i 个字符肯定和 T 的第 2 个字符 b 不相等。所以可令 next[5]=1，使 S 的第 i 个字符和 T 的第 1 个字符开始比较。这样使得模式串又向右移了一位，提高了匹配的效率。算法 4.8 是改进的求数组 next[]（在算法 4.8 中的形参是 nextval[]）的算法。

算法 4.6 是改进的模式匹配算法。它利用算法 4.7 或算法 4.8 求得的数组 next[]，避免主串指针的回溯，提高了算法的效率。algo4-1.cpp 是实现改进的模式匹配算法的程序。函数 get_next()和 get_nextval()分别求得给定的模式串的数组 next[]和 nextval[]，函数 Index_KMP()利用数组 next[]或 nextval[]求出模式串在主串中的位置。其中，next[j]=0，并不是将主串的当前字符与模式串的第 0 个字符进行比较（模式串也没有第 0 个字符），而是主串当前字符的下一个字符与模式串的第 1 个字符进行比较。

```
// algo4-1.cpp 实现算法 4.6～算法 4.8 的程序
#include"c1.h"
#include"c4-1.h" // 串的定长顺序存储结构
#include"bo4-1.cpp" // 定长顺序存储结构的基本操作(12 个)
void get_next(SString T,int next[])
{ // 求模式串 T 的 next 函数值并存入数组 next。算法 4.7
  int i = 1,j = 0;
  next[1] = 0; // T 的第 1 个字符与主串“失配”时，主串的下一字符与 T 的第 1 个字符比较(见图 4-9)
  while(i<T[0]) // 当 T[0]>1 时，next[2] = 1
    if(j == 0||T[i] == T[j]) // 初态或两字符相等
    { ++ i; // 各 + 1 继续向后比较
      ++ j;
      next[i] = j; // 主串和 T 在第 i 个字符不匹配时，前 j - 1 个字符是匹配的，只须与第 j 个字符比较
    }
    else // 两字符不等
      j = next[j]; // j 减小到前面字符相等之处
}
void get_nextval(SString T,int nextval [])
```

| next | | | | | | | |
|---|---|---|---|---|---|---|---|
| [0] | [1] | [2] | … | | | | T[0] |
| | 0 | 1 | | | | | |

图 4-9　next[]的初态

```
{ // 求模式串 T 的 next 函数修正值并存入数组 nextval。算法 4.8
  int i = 1,j = 0;
  nextval[1] = 0; // T 的第 1 个字符与主串“失配”,主串的下一字符与 T 的第 1 个字符比较(见图 4-10)
  while(i<T[0])
    if(j == 0||T[i] == T[j])
    {  ++i;
       ++j;
       if(T[i]! = T[j]) // 此处与算法 4.7 不同
         nextval[i] = j;
       else
         nextval[i] = nextval[j];
    }
    else
       j = nextval[j]; // j 减小到前面字符相等之处
}
```

nextval

| [0] | [1] | [2] | | | … | T[0] |
|---|---|---|---|---|---|---|
| | 0 | | | | | |

图 4-10　nextval[]的初态

```
int Index_KMP(SString S,SString T,int pos,int next[])
{ // 利用模式串 T 的 next 数组求 T 在主串 S 中第 pos 个字符之后的位置的 KMP 算法。
  // 其中,T 非空,1≤pos≤StrLength(S)。算法 4.6
  int i = pos,j = 1; // 初始位置
  while(i<= S[0]&&j<= T[0]) // i 和 j 分别都未超出主串 S 和模式串 T 的范围
    if(j == 0||S[i] == T[j]) // 继续比较后继字符
    {  ++i;
       ++j;
    }
    else // 模式串向右移动
       j = next[j];
  if(j>T[0]) // 匹配成功
    return i - T[0];
  else
    return 0;
}
void main()
{
  int i, * p;
  SString s1,s2; // 以教科书算法 4.8 之上的数据为例
  StrAssign(s1,"aaabaaaab"); // 由"aaabaaaab"生成主串 s1
  printf("主串为");
  StrPrint(s1); // 输出串 s1
  StrAssign(s2,"aaaab"); // 由"aaaab"生成子串 s2
  printf("子串为");
  StrPrint(s2); // 输出串 s2
  p = (int * )malloc((StrLength(s2) + 1) * sizeof(int)); // 生成 s2 的 next 数组,[0]不用
  get_next(s2,p); // 利用算法 4.7,求得 next 数组,存于 p 中
  printf("子串的 next 数组为");
  for(i = 1;i<= StrLength(s2);i ++ )
```

```
    printf("%d", *(p+i));
  printf("\n");
  i=Index_KMP(s1,s2,1,p); // 利用算法 4.6 求得串 s2 在 s1 中首次匹配的位置 i
  if(i)
    printf("主串和子串在第 %d 个字符处首次匹配\n",i);
  else
    printf("主串和子串匹配不成功\n");
  get_nextval(s2,p); // 利用算法 4.8,求得 nextval 数组,存于 p 中
  printf("子串的 nextval 数组为");
  for(i=1;i<=StrLength(s2);i++)
    printf("%d", *(p+i));
  printf("\n");
  printf("主串和子串在第 %d 个字符处首次匹配\n",Index_KMP(s1,s2,1,p));
}
```

**程序运行结果：**

```
主串为 aaabaaaab
子串为 aaaab
子串的 next 数组为 0 1 2 3 4
主串和子串在第 5 个字符处首次匹配
子串的 nextval 数组为 0 0 0 0 4
主串和子串在第 5 个字符处首次匹配
```

# 第 5 章 数组和广义表

## 5.1 数组的顺序表示和实现

```
// c5-1.h 数组的顺序存储结构。在教科书第 93 页(见图 5-1)
#define MAX_ARRAY_DIM 8 // 假设数组维数的最大值为 8
struct Array
{ ElemType * base; // 数组元素基址,由 InitArray 分配
  int dim; // 数组维数
  int * bounds; // 数组维界基址,由 InitArray 分配
  int * constants; // 数组映象函数常量基址,由 InitArray 分配
};
```

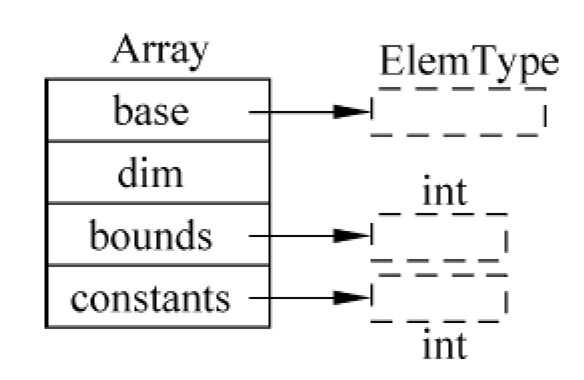


图 5-1 数组的顺序存储结构

```
// bo5-1.cpp 顺序存储数组(存储结构由 c5-1.h 定义)的基本操作(5 个)
Status InitArray(Array &A,int dim,...)
{ // 若维数 dim 和各维长度合法,则构造相应的数组 A,并返回 OK。在教科书第 93 页(见图 5-2)
  int elemtotal = 1,i; // elemtotal 是数组元素总数,初值为 1(累乘器)
  va_list ap; // 变长参数表类型,在 stdarg.h 中
  if(dim<1||dim>MAX_ARRAY_DIM) // 数组维数超出范围
    return ERROR;
  A.dim = dim; // 数组维数
  A.bounds = (int * )malloc(dim * sizeof(int)); // 动态分配数组维界基址
  if(!A.bounds)
    exit(OVERFLOW);
  va_start(ap,dim); // 变长参数“...”从形参 dim 之后开始
  for(i = 0;i<dim;++ i)
  { A.bounds[i] = va_arg(ap,int); // 逐一将变长参数赋给 A.bounds[i]
    if(A.bounds[i]<0)
      return UNDERFLOW; // 在 math.h 中定义为 4
    elemtotal * = A.bounds[i]; // 数组元素总数 = 各维长度之乘积
  }
  va_end(ap); // 结束提取变长参数
  A.base = (ElemType * )malloc(elemtotal * sizeof(ElemType)); // 动态分配数组存储空间
  if(!A.base)
```

图 5-2 A[3][4][2]数组的存储结构

```
    exit(OVERFLOW);
  A.constants = (int *)malloc(dim * sizeof(int)); // 动态分配数组偏移量基址
  if(!A.constants)
    exit(OVERFLOW);
  A.constants[dim - 1] = 1; // 最后一维的偏移量为 1
  for(i = dim - 2;i>= 0;-- i)
    A.constants[i] = A.bounds[i + 1] * A.constants[i + 1]; // 每一维的偏移量
  return OK;
}
void DestroyArray(Array &A)
{ // 销毁数组 A。在教科书第 94 页(见图 5-3)
  if(A.base) // A.base 指向存储单元
    free(A.base); // 释放 A.base 所指向的存储单元
  if(A.bounds)
    free(A.bounds);
  if(A.constants)
    free(A.constants);
  A.base = A.bounds = A.constants = NULL; // 使它们不再指向任何存储单元
  A.dim = 0;
}
```

| NULL |
|---|
| 0 |
| NULL |
| NULL |

A

图 5-3　销毁数组 A

```
Status Locate(Array A,va_list ap,int &off) // Value()、Assign()调用此函数
{ // 若 ap 指示的各下标值合法,则求出该元素在 A 中的相对地址 off。在教科书第 94 页
  int i,ind;
  off = 0;
  for(i = 0;i<A.dim;i ++ )
  { ind = va_arg(ap,int); // 逐一读取各维的下标值
    if(ind<0||ind>= A.bounds[i]) // 各维的下标值不合法
      return OVERFLOW;
    off += A.constants[i] * ind; // 相对地址 = 各维的下标值 * 本维的偏移量之和
  }
  return OK;
}
Status Value(ElemType &e,Array A,...) // 在 VC ++ 中,"..."之前的形参不能是引用类型
{ // "..."依次为各维的下标值,若各下标合法,则 e 被赋值为 A 的相应的元素值。在教科书第 94 页
  va_list ap; // 变长参数表类型,在 stdarg.h 中
  int off;
  va_start(ap,A); // 变长参数"..."从形参 A 之后开始
  if(Locate(A,ap,off) == OVERFLOW) // 调用 Locate(),求得变长参数所指单元的相对地址 off
    return ERROR;
  e = *(A.base + off); // 将变长参数所指单元的值赋给 e
  return OK;
}
Status Assign(Array A,ElemType e,...) // 变量 A 的值不变,故不需要&
{ // "..."依次为各维的下标值,若各下标合法,则将 e 的值赋给 A 的指定的元素。在教科书第 95 页
```

```
  va_list ap; // 变长参数表类型,在 stdarg.h 中
  int off;
  va_start(ap,e); // 变长参数“...”从形参 e 之后开始
  if(Locate(A,ap,off) == OVERFLOW) // 调用 Locate(),求得变长参数所指单元的相对地址 off
    return ERROR;
  *(A.base + off) = e; // 将 e 的值赋给变长参数所指单元
  return OK;
}
```

```
// main5-1.cpp 检验 bo5-1.cpp 的主程序
#include"c1.h"
typedef int ElemType; // 定义数组元素类型 ElemType 为整型
#include"c5-1.h" // 数组的动态顺序存储结构
#include"bo5-1.cpp" // 顺序数组存储结构的基本操作(5 个)
void main()
{
  Array A;
  int i,j,k,dim = 3,bound1 = 3,bound2 = 4,bound3 = 2; // A[3][4][2]数组
  ElemType e;
  InitArray(A,dim,bound1,bound2,bound3); // 构造 3×4×2 的三维数组 A(见图 5-2)
  printf("A.bounds = ");
  for(i = 0;i<dim;i++) // 顺序输出 A.bounds
    printf("%d", *(A.bounds + i));
  printf("\nA.constants = ");
  for(i = 0;i<dim;i++) // 顺序输出 A.constants
    printf("%d", *(A.constants + i));
  printf("\n%d 页 %d 行 %d 列矩阵元素如下:\n",bound1,bound2,bound3);
  for(i = 0;i<bound1;i++)
  { for(j = 0;j<bound2;j++)
    { for(k = 0;k<bound3;k++)
      { Assign(A,i * 100 + j * 10 + k,i,j,k); // 将 i×100 + j×10 + k 赋值给 A[i][j][k]
        Value(e,A,i,j,k); // 将 A[i][j][k]的值赋给 e
        printf("A[%d][%d][%d] = %2d ",i,j,k,e); // 输出 A[i][j][k]
      }
      printf("\n");
    }
    printf("\n");
  }
  printf("A.base = \n");
  for(i = 0;i<bound1 * bound2 * bound3;i++) // 顺序输出 A.base
  { printf("%4d", *(A.base + i));
    if(i % (bound2 * bound3) == bound2 * bound3 - 1)
      printf("\n");
  }
```

```
  printf("A.dim = %d\n",A.dim);
  DestroyArray(A);
}
```

**程序运行结果：**

```
A.bounds = 3 4 2
A.constants = 8 2 1
3 页 4 行 2 列矩阵元素如下：
A[0][0][0] =  0 A[0][0][1] =  1
A[0][1][0] = 10 A[0][1][1] = 11
A[0][2][0] = 20 A[0][2][1] = 21
A[0][3][0] = 30 A[0][3][1] = 31

A[1][0][0] = 100 A[1][0][1] = 101
A[1][1][0] = 110 A[1][1][1] = 111
A[1][2][0] = 120 A[1][2][1] = 121
A[1][3][0] = 130 A[1][3][1] = 131

A[2][0][0] = 200 A[2][0][1] = 201
A[2][1][0] = 210 A[2][1][1] = 211
A[2][2][0] = 220 A[2][2][1] = 221
A[2][3][0] = 230 A[2][3][1] = 231

A.base =
  0   1  10  11  20  21  30  31
100 101 110 111 120 121 130 131
200 201 210 211 220 221 230 231
A.dim = 3
```

bo5-1.cpp 中有些函数的形参有“…”，它代表变长参数表，即“…”可用若干个实参取代。这很适合含有维数不定的数组的函数。因为如果是两维数组，参数中要包括两维的长度，两个整型量；而如果是三维数组，则参数中要包括三维的长度，三个整型量。随着所构造的数组的维数不同，参数的个数也不同。这就必须使用变长参数表才能解决参数个数不定的问题。algo5-1.cpp 是采用变长参数表的一个实例。

```
// algo5-1.cpp 变长参数表(函数的实参个数可变)编程示例
#include"c1.h"
typedef int ElemType; // 定义 ElemType 为整型
ElemType Max(int num,...) // 函数功能：返回 num 个数中的最大值
{ // “...”表示变长参数表，位于形参表的最后，前面必须至少有一个固定参数
  va_list ap; // 定义 ap 是变长参数表类型(C 语言的数据类型)，在 stdarg.h 中
  int i;
  ElemType m,n;
  if(num<1)
    exit(OVERFLOW);
```

```
  va_start(ap,num); // ap 指向固定参数 num 后面的实参表
  m = va_arg(ap,ElemType); // 读取 ap 所指的实参,其类型为 ElemType,将其赋给 m,ap 向后移
  for(i = 1;i<num;++ i) // 从第 2 个数到最后一个数
  { n = va_arg(ap,ElemType); // 依次读取 ap 所指的实参,将其赋给 n,ap 向后移
    if(m<n)
      m = n; // m 中存放最大值
  }
  va_end(ap); // 与 va_start()配对,结束对变长参数表的读取,ap 不再指向变长参数表
  return m; // 将最大值返回
}
void main()
{
  printf("1.最大值为 %d\n",Max(4,7,9,5,8)); // 在 4 个数中求最大值,ap 最初指向 7
  printf("2.最大值为 %d\n",Max(3,17,36,25)); // 在 3 个数中求最大值,ap 最初指向 17
}
```

**程序运行结果:**

```
1.最大值为 9
2.最大值为 36
```

其实,printf()就是含有变长参数表的 C 语言库函数,它的第 1 个形参是字符串常量或字符型指针,第 2 个形参是变长参数表。因此,我们才可以在 1 个 printf()函数中输出任意个变量。

## 5.2　矩阵的压缩存储

```
// c5-2.h 稀疏矩阵的三元组顺序表存储结构。在教科书第 98 页
#define MAX_SIZE 100 // 非零元个数的最大值(见图 5-4)
struct Triple
{ int i,j; // 行下标,列下标
  ElemType e; // 非零元素值
};
struct TSMatrix
{ Triple data[MAX_SIZE + 1]; // 非零元三元组表,data[0]未用
  int mu,nu,tu; // 矩阵的行数,列数,非零元个数
};
```

图 5-5 是采用三元组顺序表存储稀疏矩阵的例子。为简化算法,在创建稀疏矩阵输入非零元时,要按行、列的顺序由小到大输入。

```
// func5-1.cpp
int comp(int c1,int c2) // 比较整数 c1 和 c2 的大小关系,分别返回 - 1、0 和 1
{ // AddSMatrix()和 MultSMatrix()函数要用到
  if(c1<c2)
```

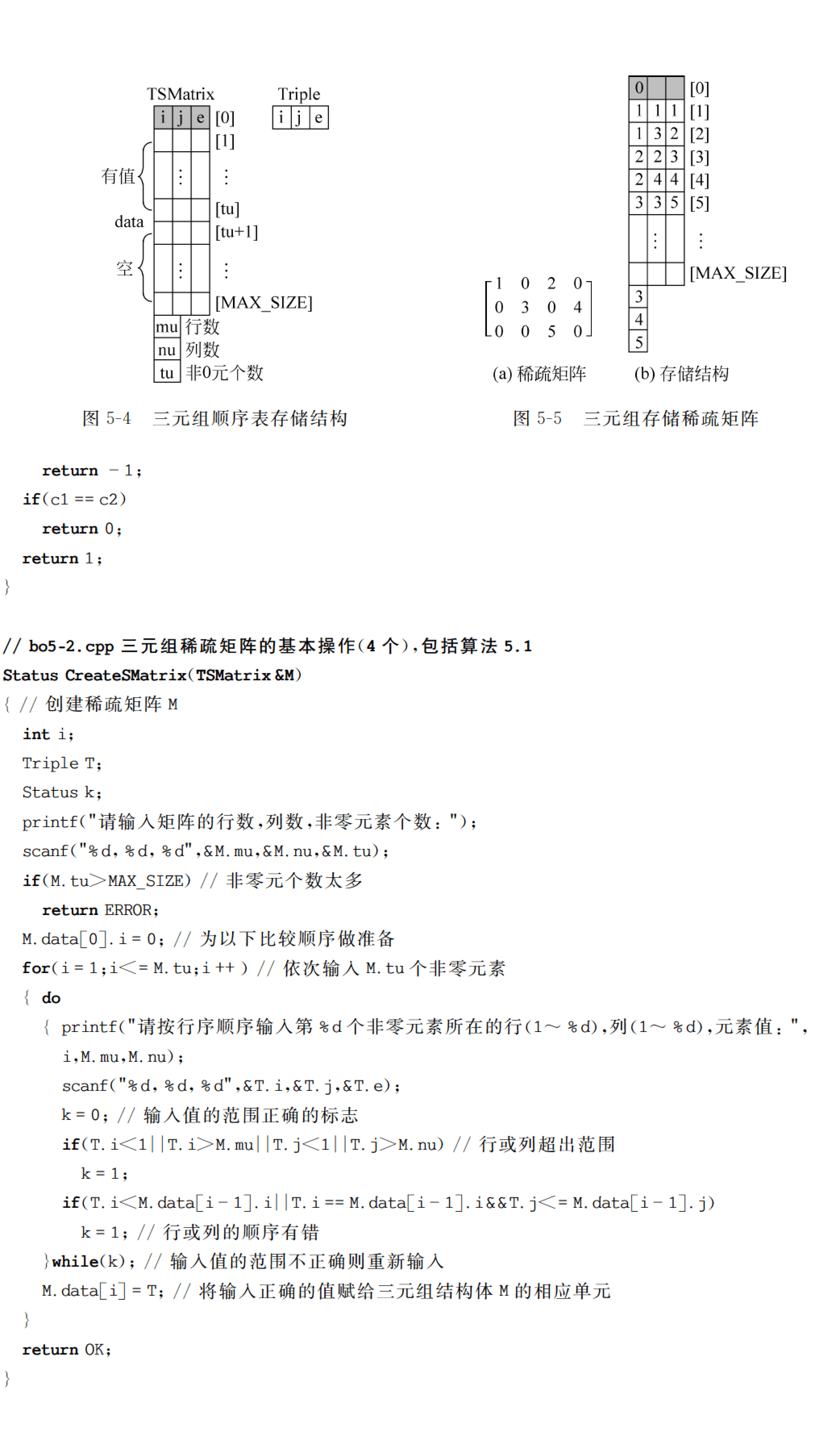



图 5-4　三元组顺序表存储结构

图 5-5　三元组存储稀疏矩阵

```
    return -1;
  if(c1 == c2)
    return 0;
  return 1;
}
```

```
// bo5-2.cpp 三元组稀疏矩阵的基本操作(4 个),包括算法 5.1
Status CreateSMatrix(TSMatrix &M)
{ // 创建稀疏矩阵 M
  int i;
  Triple T;
  Status k;
  printf("请输入矩阵的行数,列数,非零元素个数:");
  scanf("%d,%d,%d",&M.mu,&M.nu,&M.tu);
  if(M.tu>MAX_SIZE) // 非零元个数太多
    return ERROR;
  M.data[0].i = 0; // 为以下比较顺序做准备
  for(i = 1;i<= M.tu;i ++ ) // 依次输入 M.tu 个非零元素
  { do
    { printf("请按行序顺序输入第%d个非零元素所在的行(1～%d),列(1～%d),元素值:",
      i,M.mu,M.nu);
      scanf("%d,%d,%d",&T.i,&T.j,&T.e);
      k = 0; // 输入值的范围正确的标志
      if(T.i<1||T.i>M.mu||T.j<1||T.j>M.nu) // 行或列超出范围
        k = 1;
      if(T.i<M.data[i-1].i||T.i == M.data[i-1].i&&T.j<= M.data[i-1].j)
        k = 1; // 行或列的顺序有错
    }while(k); // 输入值的范围不正确则重新输入
    M.data[i] = T; // 将输入正确的值赋给三元组结构体 M 的相应单元
  }
  return OK;
}
```

```
Status AddSMatrix(TSMatrix M,TSMatrix N,TSMatrix &Q)
{ // 求稀疏矩阵的和 Q = M + N
  int m = 1,n = 1,q = 0;
  if(M.mu! = N.mu||M.nu! = N.nu) // M、N 两稀疏矩阵行或列数不同
    return ERROR;
  Q.mu = M.mu; // 设置稀疏矩阵 Q 的行数和列数
  Q.nu = M.nu;
  while(m<= M.tu&&n<= N.tu) // 矩阵 M 和 N 的元素都未处理完
    switch(comp(M.data[m].i,N.data[n].i)) // 比较两当前元素的行值关系
    { case  - 1:Q.data[ ++ q] = M.data[m ++ ]; // 矩阵 M 的行值小,将 M 的当前元素值赋给矩阵 Q
                break;
      case  0:switch(comp(M.data[m].j,N.data[n].j))
                { // M、N 矩阵当前元素的行值相等,继续比较两当前元素的列值关系
                  case  - 1:Q.data[ ++ q] = M.data[m ++ ]; // 矩阵 M 的列值小,将 M 的值赋给矩阵 Q
                          break;
                  case  0: // M、N 矩阵当前非零元素的行列均相等,将两元素值求和并赋给矩阵 Q
                          Q.data[ ++ q] = M.data[m ++ ];
                          Q.data[q].e += N.data[n ++ ].e;
                          if(Q.data[q].e == 0) // 两元素值之和为 0,不存入稀疏矩阵
                            q --;
                          break;
                  case  1:Q.data[ ++ q] = N.data[n ++ ]; // 矩阵 N 的列值小,将 N 的值赋给矩阵 Q
                }
                break;
      case  1:Q.data[ ++ q] = N.data[n ++ ]; // 矩阵 N 的行值小,将 N 的当前元素值赋给矩阵 Q
    } // 以下 2 个循环最多执行 1 个
  while(m<= M.tu) // 矩阵 N 的元素已全部处理完毕,处理矩阵 M 的元素
    Q.data[ ++ q] = M.data[m ++ ];
  while(n<= N.tu) // 矩阵 M 的元素已全部处理完毕,处理矩阵 N 的元素
    Q.data[ ++ q] = N.data[n ++ ];
  if(q>MAX_SIZE) // 非零元素个数太多
    return ERROR;
  Q.tu = q; // 矩阵 Q 的非零元素个数
  return OK;
}
void TransposeSMatrix(TSMatrix M,TSMatrix &T)
{ // 求稀疏矩阵 M 的转置矩阵 T。修改算法 5.1
  int p,col,q = 1; // q 指示转置矩阵 T 的当前元素,初值为 1
  T.mu = M.nu; // 矩阵 T 的行数 = 矩阵 M 的列数
  T.nu = M.mu; // 矩阵 T 的列数 = 矩阵 M 的行数
  T.tu = M.tu; // 矩阵 T 的非零元素个数 = 矩阵 M 的非零元素个数
  if(T.tu) // 矩阵非空
    for(col = 1;col<= M.nu;++ col) // 从矩阵 T 的第 1 行到最后一行
      for(p = 1;p<= M.tu;++ p) // 对于矩阵 M 的所有元素
        if(M.data[p].j == col) // 该元素的列数 = 当前矩阵 T 的行数
```

```
        { T.data[q].i = M.data[p].j; // 将矩阵 M 的值行列对调赋给 T 的当前元素
          T.data[q].j = M.data[p].i;
          T.data[q ++ ].e = M.data[p].e; // 转置矩阵 T 的当前元素指针 + 1
        }
}
Status MultSMatrix(TSMatrix M,TSMatrix N,TSMatrix &Q)
{ // 求稀疏矩阵的乘积 Q = M × N
  int i,j,q,p;
  ElemType Qs; // 矩阵单元 Q[i][j]的临时存放处
  TSMatrix T; // N 的转秩矩阵
  if(M.nu! = N.mu) // 矩阵 M 和 N 无法相乘
    return ERROR;
  Q.mu = M.mu; // Q 的行数 = M 的行数
  Q.nu = N.nu; // Q 的列数 = N 的列数
  Q.tu = 0; // Q 的非零元素个数的初值为 0
  TransposeSMatrix(N,T); // T 是 N 的转秩矩阵
  for(i = 1;i<= Q.mu;i ++ ) // 对于 M 的每一行,求 Q[i][]
  { q = 1; // q 指向 T 的第 1 个非零元素
    for(j = 1;j<= T.mu;j ++ ) // 对于 T 的每一行(即 N 的每一列),求 Q[i][j]
    { Qs = 0; // 设置 Q[i][j]的初值为 0
      p = 1; // p 指向 M 的第 1 个非零元素
      while(M.data[p].i<i) // 使 p 指向矩阵 M 的第 i 行的第 1 个非零元素
        p ++;
      while(T.data[q].i<j) // 使 q 指向矩阵 T 的第 j 行(即矩阵 N 的第 j 列)的第 1 个非零元素
        q ++;
      while(p<= M.tu&&q<= T.tu&&M.data[p].i == i&&T.data[q].i == j)
      // [p]仍是 M 的第 i 行的非零元素且[q]仍是 T 的第 j 行(即 N 的第 j 列)的非零元素
        switch(comp(M.data[p].j,T.data[q].j))
        { // 比较 M 矩阵当前元素的列值和 T 矩阵当前元素的列值(即 N 矩阵当前元素的行值)
          case  - 1:p ++; // M 矩阵当前元素的列值<T(N)矩阵当前元素的列(行)值,p 向后移
                 break;
          // M 当前元素的列值 = T(N)当前元素的列(行)值,则两值相乘并累加到 Qs,p、q 均向后移
          case   0:Qs += M.data[p ++ ].e * T.data[q ++ ].e;
                 break;
          case   1:q ++; // M 矩阵当前元素的列值>T(N)矩阵当前元素的列(行)值,q 向后移
        }
      if(Qs) // Q[i][j]不为 0
      { if( ++ Q.tu>MAX_SIZE) // Q 的非零元素个数 + 1,如果非零元素个数太多
          return ERROR;
        Q.data[Q.tu].i = i; // 将 Q[i][j]按顺序存入稀疏矩阵 Q
        Q.data[Q.tu].j = j;
        Q.data[Q.tu].e = Qs;
      }
    }
  }
```

```
    return OK;
}
```

```
// bo5-3.cpp 三元组稀疏矩阵的基本操作(4 个),也可用于行逻辑链接结构
void DestroySMatrix(TSMatrix &M)
{ // 销毁稀疏矩阵 M(见图 5-6)
    M.mu = M.nu = M.tu = 0;
}
void PrintSMatrix(TSMatrix M)
{ // 按矩阵形式输出 M
    int i,j,k = 1; // 非零元计数器,初值为 1
    Triple *p = M.data + 1; // p 指向 M 的第 1 个非零元素
    for(i = 1;i<= M.mu;i ++ ) // 从第 1 行到最后一行
    { for(j = 1;j<= M.nu;j ++ ) // 从第 1 列到最后一列
        if(k<= M.tu&&p->i == i&&p->j == j) // p 指向非零元,且 p 所指元素为当前循环在处理元素
        { printf("%3d",(p ++ )->e); // 输出 p 所指元素的值,p 指向下一个元素
          k ++; // 计数器 + 1
        }
        else // p 所指元素不是当前循环在处理元素
          printf("%3d",0); // 输出 0
      printf("\n");
    }
}
void CopySMatrix(TSMatrix M,TSMatrix &T)
{ // 由稀疏矩阵 M 复制得到 T
    T = M;
}
Status SubtSMatrix(TSMatrix M,TSMatrix N,TSMatrix &Q)
{ // 求稀疏矩阵的差 Q = M - N
    int i;
    if(M.mu! = N.mu||M.nu! = N.nu) // M、N 两稀疏矩阵行或列数不同
      return ERROR;
    for(i = 1;i<= N.tu;++ i) // 对于 N 的每一元素,其值乘以 - 1
      N.data[i].e *=- 1;
    AddSMatrix(M,N,Q); // Q = M + ( - N)
    return OK;
}
```


图 5-6　销毁稀疏矩阵 M

```
// func5-2.cpp 稀疏矩阵的主函数,main5-2.cpp 和 main5-3.cpp 调用
void main()
{
    TSMatrix A,B,C;
    printf("创建矩阵 A: ");
    CreateSMatrix(A); // 创建矩阵 A
```

```
    PrintSMatrix(A); // 输出矩阵 A
    CopySMatrix(A,B); // 由矩阵 A 复制矩阵 B
    printf("由矩阵 A 复制矩阵 B:\n");
    PrintSMatrix(B); // 输出矩阵 B
    DestroySMatrix(B); // 销毁矩阵 B
    printf("销毁矩阵 B 后:\n");
    PrintSMatrix(B); // 输出矩阵 B
    printf("创建矩阵 B2:(与矩阵 A 的行、列数相同,行、列分别为%d,%d)\n",A.mu,A.nu);
    CreateSMatrix(B); // 创建矩阵 B
    PrintSMatrix(B); // 输出矩阵 B
    AddSMatrix(A,B,C); // 矩阵相加,C=A+B
    printf("矩阵 C1(A+B):\n");
    PrintSMatrix(C); // 输出矩阵 C
    SubtSMatrix(A,B,C); // 矩阵相减,C=A-B
    printf("矩阵 C2(A-B):\n");
    PrintSMatrix(C); // 输出矩阵 C
    TransposeSMatrix(A,C); // 矩阵 C 是矩阵 A 的转秩矩阵
    printf("矩阵 C3(A 的转置):\n");
    PrintSMatrix(C); // 输出矩阵 C
    printf("创建矩阵 A2:");
    CreateSMatrix(A); // 创建矩阵 A
    PrintSMatrix(A); // 输出矩阵 A
    printf("创建矩阵 B3:(行数应与矩阵 A2 的列数相同=%d)\n",A.nu);
    CreateSMatrix(B); // 创建矩阵 B
    PrintSMatrix(B); // 输出矩阵 B
  #ifndef FLAG // 未定义 FLAG
    MultSMatrix(A,B,C); // 矩阵相乘,C=A×B
  #else // 定义了 FLAG
    MultSMatrix1(A,B,C); // 另一个矩阵相乘函数 C=A×B,在 bo5-3.cpp 中
  #endif
    printf("矩阵 C5(A×B):\n");
    PrintSMatrix(C); // 输出矩阵 C
  }
```

```
// main5-2.cpp 检验 bo5-2.cpp 和 bo5-3.cpp 的主程序
#include"c1.h"
typedef int ElemType; // 定义矩阵元素类型 ElemType 为整型
#include"c5-2.h" // 稀疏矩阵的三元组顺序表存储结构
#include"func5-1.cpp" // comp()函数
#include"bo5-2.cpp" // 三元组稀疏矩阵的基本操作(4 个)
#include"bo5-3.cpp" // 也可用于行逻辑链接结构三元组稀疏矩阵的基本操作(4 个)
#include"func5-2.cpp" // 主函数
```

**程序运行结果：**

```
创建矩阵 A：请输入矩阵的行数，列数，非零元素个数：3,3,2↙
请按行序顺序输入第 1 个非零元素所在的行(1～3)，列(1～3)，元素值：1,2,1↙
请按行序顺序输入第 2 个非零元素所在的行(1～3)，列(1～3)，元素值：2,2,2↙
  0  1  0（见图 5-7）
  0  2  0
  0  0  0
由矩阵 A 复制矩阵 B：
  0  1  0
  0  2  0
  0  0  0
销毁矩阵 B 后：
创建矩阵 B2：(与矩阵 A 的行、列数相同，行、列分别为 3,3)
请输入矩阵的行数，列数，非零元素个数：3,3,1↙
请按行序顺序输入第 1 个非零元素所在的行(1～3)，列(1～3)，元素值：1,2,1↙
  0  1  0
  0  0  0
  0  0  0
矩阵 C1(A＋B)：
  0  2  0
  0  2  0
  0  0  0
矩阵 C2(A－B)：
  0  0  0
  0  2  0
  0  0  0
矩阵 C3(A 的转置)：
  0  0  0
  1  2  0
  0  0  0
创建矩阵 A2：请输入矩阵的行数，列数，非零元素个数：2,3,2↙
请按行序顺序输入第 1 个非零元素所在的行(1～2)，列(1～3)，元素值：1,1,1↙
请按行序顺序输入第 2 个非零元素所在的行(1～2)，列(1～3)，元素值：2,3,2↙
  1  0  0
  0  0  2
创建矩阵 B3：(行数应与矩阵 A2 的列数相同＝3)
请输入矩阵的行数，列数，非零元素个数：3,2,2↙
请按行序顺序输入第 1 个非零元素所在的行(1～3)，列(1～2)，元素值：2,2,1↙
请按行序顺序输入第 2 个非零元素所在的行(1～3)，列(1～2)，元素值：3,1,2↙
  0  0
  0  1
  2  0
矩阵 C5(A×B)：
  0  0
  4  0
```

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

图 5-7　矩阵 A

```
// algo5-2.cpp 实现算法 5.2 的程序
#include"c1.h"
typedef int ElemType; // 定义矩阵元素类型 ElemType 为整型
#include"c5-2.h" // 稀疏矩阵的三元组顺序表存储结构
#include"func5-1.cpp" // comp()函数
#include"bo5-2.cpp" // 三元组稀疏矩阵的基本操作(4 个)
#include"bo5-3.cpp" // 也可用于行逻辑链接结构三元组稀疏矩阵的基本操作(4 个)
void FastTransposeSMatrix(TSMatrix M,TSMatrix &T)
{ // 快速求稀疏矩阵 M 的转置矩阵 T。修改算法 5.2
  int p,q,col,*num,*cpot;
  num=(int*)malloc((M.nu+1)*sizeof(int)); // 存 M 每列(T 每行)非零元素个数([0]不用)
  cpot=(int*)malloc((M.nu+1)*sizeof(int)); // 存 T 每行下一个非零元素的位置([0]不用)
  T.mu=M.nu; // T 的行数=M 的列数
  T.nu=M.mu; // T 的列数=M 的行数
  T.tu=M.tu; // T 的非零元素个数=M 的非零元素个数
  if(T.tu) // T 是非零矩阵
  { for(col=1;col<=M.nu;++col) // 从 M 的第 1 列到最后一列
      num[col]=0; // 计数器初值设为 0
    for(p=1;p<=M.tu;++p) // 对于 M 的每一个非零元素
      ++num[M.data[p].j]; // 根据它所在的列进行统计
    cpot[1]=1; // T 的第 1 行的第 1 个非零元在 T.data 中的序号为 1
    for(col=2;col<=M.nu;++col) // 从 M(T)的第 2 列(行)到最后一列(行)
      cpot[col]=cpot[col-1]+num[col-1]; // 求 T 的第 col 行第 1 个非零元在 T.data 中的序号
    for(p=1;p<=M.tu;++p) // 对于 M 的每一个非零元素
    { col=M.data[p].j; // 将其在 M 中的列数赋给 col
      q=cpot[col]; // q 指示 M 当前的元素在 T 中的序号
      T.data[q].i=M.data[p].j; // 将 M 当前的元素转秩赋给 T
      T.data[q].j=M.data[p].i;
      T.data[q].e=M.data[p].e;
      ++cpot[col]; // T 第 col 行的下 1 个非零元在 T.data 中的序号比当前元素的序号大 1
    }
  }
  free(num); // 释放 num 和 cpot 所指向的动态存储空间
  free(cpot);
}
void main()
{
  TSMatrix A,B;
  printf("创建矩阵 A: ");
  CreateSMatrix(A); // 创建矩阵 A
  PrintSMatrix(A); // 输出矩阵 A
  FastTransposeSMatrix(A,B); // B 是 A 的转秩矩阵
  printf("矩阵 B(A 的快速转置): \n");
  PrintSMatrix(B); // 输出矩阵 B
}
```

**程序运行结果：**

```
创建矩阵 A：请输入矩阵的行数，列数，非零元素个数：3,2,4↙
请按行序顺序输入第 1 个非零元素所在的行(1～3)，列(1～2)，元素值：1,1,1↙
请按行序顺序输入第 2 个非零元素所在的行(1～3)，列(1～2)，元素值：2,1,2↙
请按行序顺序输入第 3 个非零元素所在的行(1～3)，列(1～2)，元素值：3,1,3↙
请按行序顺序输入第 4 个非零元素所在的行(1～3)，列(1～2)，元素值：3,2,4↙
  1  0
  2  0
  3  4
矩阵 B(A 的快速转置)：
  1  2  3
  0  0  4
```

由于稀疏矩阵的三元组顺序表存储结构要求先按行、同行再按列顺序存储非零元素，算法 5.1(在 bo5-2.cpp 中)采用了双重循环求转置矩阵，对于外层循环 col(列)的每一个值，对所有的非零元素，如果其列数与 col 相等，则按顺序存入转秩矩阵。这就保证了转秩矩阵也是先按行、同行再按列顺序存储非零元素。所以它的时间复杂度为 O(列数×非零元素个数)。而算法 5.2(在 algo5-2.cpp 中)采用了 2 个单循环。第 1 个循环，对所有的非零元素，计算其所在列并计数，得到每列(col)的非零元素个数 num[col]及每列第 1 个非零元素在转秩矩阵中的存储位置 cpot[col]。第 2 个循环，对所有的非零元素，根据其列数和 cpot[col]的当前值，存入转秩矩阵。由于还要给 num[col]和 cpot[col]赋初值，它的时间复杂度为 O(列数＋非零元素个数)。

```
// c5-3.h 稀疏矩阵的三元组行逻辑链接的顺序表存储结构(见图 5-8)
#define MAX_SIZE 100 // 非零元个数的最大值。在教科书第 100 页
#define MAX_RC 20 // 最大行数
struct Triple // 同 c5-2.h
{ int i,j; // 行下标,列下标
  ElemType e; // 非零元素值
};
struct RLSMatrix
{ Triple data[MAX_SIZE+1]; // 非零元三元组表,data[0]未用
  int rpos[MAX_RC+1]; // 各行第 1 个非零元素的位置表,比 c5-2.h 增加的项
  int mu,nu,tu; // 矩阵的行数,列数,非零元个数
};
```

三元组行逻辑链接的顺序表存储结构(c5-3.h)只是比三元组顺序表存储结构(c5-2.h)增加了 rpos 数组，用以存放各行的第一个非零元素在 data 数组中的位置。方便查找指定行的元素。图 5-9 是采用三元组行逻辑链接的顺序表存储稀疏矩阵的实例，由 rpos[2]和 rpos[3]的值可知：第 2 行的非零元素有 2 个，位于 data[3]和 data[4]；data[5]是第 3 行的第一个非零元素。由于 c5-3.h 存储结构和 c5-2.h 存储结构有很多共同之处，c5-3.h 存储结构的一些基本操作和 c5-2.h 存储结构的基本操作(在 bo5-3.cpp 中)一样，只要通过宏定义将 TSMatrix 类型定义成 RLSMatrix 类型即可。和 c5-2.h 存储结构一样，c5-3.h 存储结构在

创建稀疏矩阵输入非零元时,也要按行、列的顺序由小到大输入。

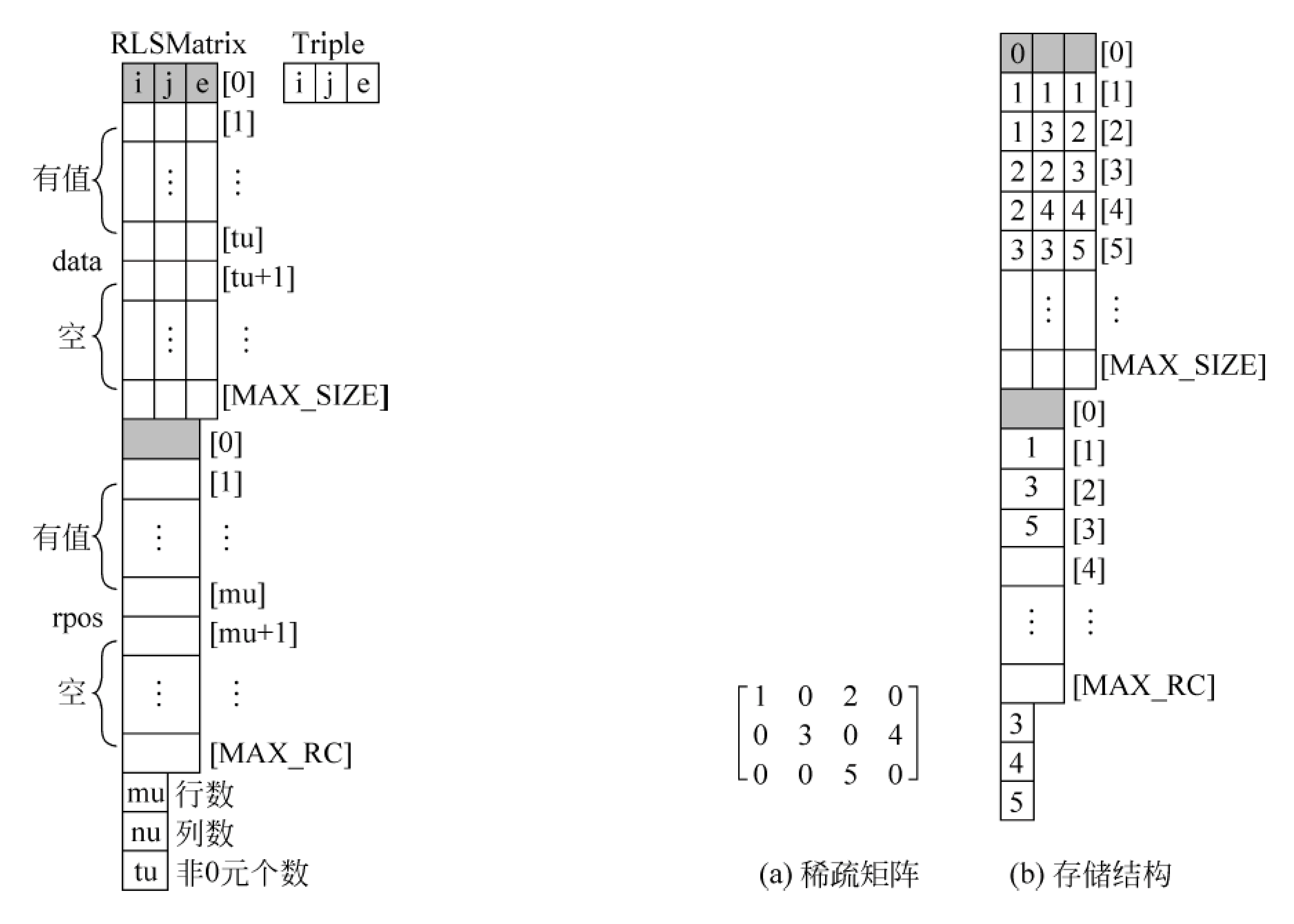



图 5-8 三元组行逻辑链接顺序表存储结构　　图 5-9 采用三元组行逻辑链接存储稀疏矩阵的示例

```
// bo5-4.cpp 行逻辑链接稀疏矩阵(存储结构由 c5-3.h 定义)的基本操作(4 个),包括算法 5.3
Status CreateSMatrix(RLSMatrix &M)
{ // 创建稀疏矩阵 M
  int i,j;
  Triple T;
  Status k;
  printf("请输入矩阵的行数,列数,非零元素个数:");
  scanf("%d,%d,%d",&M.mu,&M.nu,&M.tu);
  if(M.tu>MAX_SIZE||M.mu>MAX_RC) // 矩阵 M 的非零元个数太多或行数太多
    return ERROR;
  M.data[0].i=0; // 为以下比较做准备
  for(i=1;i<=M.tu;i++) // 依次输入 M.tu 个非零元素
  { do
    { printf("请按行序顺序输入第%d 个非零元素所在的行(1～%d),列(1～%d),元素值:",
      i,M.mu,M.nu);
      scanf("%d,%d,%d",&T.i,&T.j,&T.e);
      k=0; // 输入值的范围正确的标志
      if(T.i<1||T.i>M.mu||T.j<1||T.j>M.nu) // 行或列超出范围
        k=1;
      if(T.i<M.data[i-1].i||T.i==M.data[i-1].i&&T.j<=M.data[i-1].j)
        k=1; // 行或列的顺序有错
    }while(k); // 输入值的范围不正确则重新输入
    M.data[i]=T; // 将输入正确的值赋给 M 的相应存储单元
  }
```

```
  for(i=1;i<=M.mu;i++) // 给 rpos[]赋初值 1(每行第 1 个非零元素的初始位置)
    M.rpos[i]=1;
  for(i=1;i<=M.tu;i++) // 对于每个非零元素,按行统计,并记入 rpos[]
    for(j=M.data[i].i+1;j<=M.mu;j++) // 从非零元素所在行的下一行起
      M.rpos[j]++; // 每行第 1 个非零元素的位置+1
  return OK;
}
Status AddSMatrix(RLSMatrix M,RLSMatrix N,RLSMatrix &Q)
{ // 求稀疏矩阵的和 Q=M+N
  int k,p,q,up,uq;
  if(M.mu!=N.mu||M.nu!=N.nu) // M、N 两稀疏矩阵行或列数不同
    return ERROR;
  Q.mu=M.mu; // 设置稀疏矩阵 Q 的行数和列数
  Q.nu=M.nu;
  Q.tu=0; // 矩阵 Q 非零元素个数初值
  for(k=1;k<=M.mu;++k) // 对于每一行,k 指示行号
  { Q.rpos[k]=Q.tu+1; // 矩阵 Q 第 k 行的第 1 个元素的位置
    p=M.rpos[k]; // p 指示矩阵 M 第 k 行当前元素的序号
    q=N.rpos[k]; // q 指示矩阵 N 第 k 行当前元素的序号
    if(k<M.mu) // 不是最后一行
    { up=M.rpos[k+1]; // 下一行的第 1 个元素的位置是本行元素的上界
      uq=N.rpos[k+1];
    }
    else // 是最后一行
    { up=M.tu+1; // 给最后 1 行设上界
      uq=N.tu+1;
    }
    while(p<up&&q<uq) // 矩阵 M、N 均有第 k 行元素未处理
      switch(comp(M.data[p].j,N.data[q].j)) // 比较两当前元素的列值关系
      { // 矩阵 M 当前元素的列<矩阵 N 当前元素的列,将 M 的当前元素值赋给矩阵 Q,p 向后移
        case -1:Q.data[++Q.tu]=M.data[p++];
                break;
        case  0:if(M.data[p].e+N.data[q].e!=0)
                { // 矩阵 M 当前元素的列=矩阵 N 当前元素的列,如果和不为 0
                  Q.data[++Q.tu]=M.data[p]; // 将 M 的当前元素值赋给矩阵 Q
                  Q.data[Q.tu].e+=N.data[q].e; // 将 N 的当前元素值中成员 e 的值加入其中
                }
                p++; // p、q 均向后移,无论和是否为 0
                q++;
                break;
        // 矩阵 M 当前元素的列>矩阵 N 当前元素的列,将 N 的当前元素值赋给矩阵 Q,q 向后移
        case  1:Q.data[++Q.tu]=N.data[q++];
      } // 以下 2 个循环最多执行 1 个
    while(p<M.rpos[k+1]&&p<=M.tu) // N 的第 k 行元素已全部处理,M 还有第 k 行的元素未处理
      Q.data[++Q.tu]=M.data[p++]; // 将 M 的当前值赋给 Q,p 向后移
```

```
      while(q<N.rpos[k+1]&&q<=N.tu) // M的第k行元素已全部处理,N还有第k行的元素未处理
        Q.data[++Q.tu]=N.data[q++]; // 将N的当前值赋给Q,q向后移
    }
    if(Q.tu>MAX_SIZE) // 非零元素个数太多
      return ERROR;
    else
      return OK;
}
void TransposeSMatrix(RLSMatrix M,RLSMatrix &T)
{ // 求稀疏矩阵M的转置矩阵T
  int i,j,k,num[MAX_RC+1]; // [0]不用
  T.mu=M.nu; // 矩阵T的行数=矩阵M的列数
  T.nu=M.mu; // 矩阵T的列数=矩阵M的行数
  T.tu=M.tu; // 矩阵T的非零元素个数=矩阵M的非零元素个数
  if(T.tu) // 矩阵非空
  { for(i=1;i<=T.mu;++i) // 从矩阵T的第1行到最后一行
      num[i]=0; // 矩阵T每行非零元素个数,初值设置为0
    for(i=1;i<=M.tu;++i) // 对于M中的每一个非零元素,按列统计
      ++num[M.data[i].j]; // num[]=T的每行(M的每列)非零元素个数
    T.rpos[1]=1; // 矩阵T中第1行的第1个非零元素的序号是1
    for(i=2;i<=T.mu;++i) // 从矩阵T的第2行到最后一行
      T.rpos[i]=T.rpos[i-1]+num[i-1]; // 求T中第i行的第1个非零元素的序号
    for(i=1;i<=T.mu;++i)
      num[i]=T.rpos[i]; // num[]=M的当前非零元素在T中应存放的位置
    for(i=1;i<=M.tu;++i) // 对于M中的每一个非零元素
    { j=M.data[i].j; // 在矩阵T中的行数
      k=num[j]++; // 在矩阵T中的序号,num[j]+1
      T.data[k].i=M.data[i].j; // 将M.data[i]行列对调赋给T.data[k]
      T.data[k].j=M.data[i].i;
      T.data[k].e=M.data[i].e;
    }
  }
}
Status MultSMatrix(RLSMatrix M,RLSMatrix N,RLSMatrix &Q)
{ // 求稀疏矩阵乘积Q=M×N。算法5.3
  int arow,brow,p,q,ccol,ctemp[MAX_RC+1],t,tp;
  if(M.nu!=N.mu) // 矩阵M和N无法相乘
    return ERROR;
  Q.mu=M.mu; // Q的行数=M的行数
  Q.nu=N.nu; // Q的列数=N的列数
  Q.tu=0; // Q的非零元素个数的初值为0
  if(M.tu*N.tu!=0) // Q是非零矩阵
  for(arow=1;arow<=M.mu;++arow) // 对M的每一行,arow是M的当前行
  { for(ccol=1;ccol<=Q.nu;++ccol) // 从Q的第1列到最后一列
      ctemp[ccol]=0; // Q的当前行的各列元素累加器清零
```

```
    Q.rpos[arow] = Q.tu + 1; // Q 当前行的第 1 个元素位于上一行最后 1 个元素之后
    if(arow<M.mu) // 不是最后一行
      tp = M.rpos[arow + 1]; // 下一行的第 1 个元素的位置是本行元素的上界
    else // 是最后一行
      tp = M.tu + 1; // 给最后一行设上界
    for(p = M.rpos[arow];p<tp;++ p) // 对 M 当前行中每一个非零元
    { brow = M.data[p].j; // 找到对应元在 N 中的行号(M 当前元的列号)
      if(brow<N.mu) // 不是最后一行
        t = N.rpos[brow + 1]; // 下一行的第 1 个元素的位置是本行元素的上界
      else // 不是最后一行
        t = N.tu + 1; // 给最后一行设上界
      for(q = N.rpos[brow];q<t;++ q) // 对 N 当前行中每一个非零元
      { ccol = N.data[q].j; // 乘积元素在 Q 中的列号
        ctemp[ccol] += M.data[p].e * N.data[q].e; // 将乘积累加到 Q 的 arow 行 ccol 列中
      }
    } // 求得 Q 中第 arow 行的所有列的元素值,存于 ctemp[]中
    for(ccol = 1;ccol<= Q.nu;++ ccol) // 对于第 arow 行的所有列,只存储其中的非零元
      if(ctemp[ccol]) // 该列的值不为 0
      { if( ++ Q.tu>MAX_SIZE) // Q 的非零元素个数 + 1,如果非零元个数太多
          return ERROR;
        Q.data[Q.tu].i = arow; // 将 Q[i][j]按顺序存入稀疏矩阵 Q
        Q.data[Q.tu].j = ccol;
        Q.data[Q.tu].e = ctemp[ccol];
      }
  }
  return OK;
}
Status MultSMatrix1(RLSMatrix M,RLSMatrix N,RLSMatrix &Q)
{ // 另一种求稀疏矩阵乘积 Q = M × N 的方法(不使用临时数组,利用 N 的转秩矩阵 T)
  int i,j,q,p,up,uq;
  ElemType Qs; // 矩阵单元 Q[i][j]的临时存放处
  RLSMatrix T; // N 的转秩矩阵
  if(M.nu! = N.mu) // 矩阵 M 和 N 无法相乘
    return ERROR;
  Q.mu = M.mu; // Q 的行数 = M 的行数
  Q.nu = N.nu; // Q 的列数 = N 的列数
  Q.tu = 0; // Q 的非零元素个数的初值为 0
  TransposeSMatrix(N,T); // T 是 N 的转秩矩阵
  for(i = 1;i<= Q.mu;i ++ ) // 对于 Q 的每一行
    for(j = 1;j<= Q.nu;j ++ ) // 对于 Q 的每一列,求 Q[i][j]
    { Qs = 0; // Q[i][j]的初值为 0
      p = M.rpos[i]; // p 指示矩阵 M 在 i 行的第 1 个非零元素的位置
      q = T.rpos[j]; // q 指示矩阵 T 在 j 行(N 在 j 列)的第 1 个非零元素的位置
      if(i<M.mu) // 不是最后一行
        up = M.rpos[i + 1]; // 下一行的第 1 个元素的位置是本行元素的上界
      else // 是最后一行
        up = M.tu + 1; // 给最后一行设上界
```

```
            if(j<T.mu) // 不是最后一行
              uq = T.rpos[j + 1]; // 下一行的第 1 个元素的位置是本行元素的上界
            else // 是最后一行
              uq = T.tu + 1; // 给最后一行设上界
            while(p<up&&q<uq) // p、q 分别指示矩阵 M、T 中第 i、j 行元素
              switch(comp(M.data[p].j,T.data[q].j))
              { // 比较 M 矩阵当前元素的列值和 T 矩阵当前元素的列值(即 N 矩阵当前元素的行值)
                case -1: p ++; // M 矩阵当前元素的列值<T(N)矩阵当前元素的列(行)值,p 向后移
                         break;
                // M 当前元素的列值 = T(N)当前元素的列(行)值,则两值相乘并累加到 Qs,p、q 均向后移
                case  0: Qs += M.data[p ++ ].e * T.data[q ++ ].e;
                         break;
                case  1: q ++; // M 矩阵当前元素的列值>T(N)矩阵当前元素的列(行)值,q 向后移
              }
            if(Qs) // Q[i][j]不为 0
            { if( ++ Q.tu>MAX_SIZE) // Q 的非零元素个数 + 1,如果非零元个数太多
                return ERROR;
              Q.data[Q.tu].i = i; // 将 Q[i][j]按顺序存入稀疏矩阵 Q
              Q.data[Q.tu].j = j;
              Q.data[Q.tu].e = Qs;
            }
          }
    return OK;
  }
```

```
// main5-3.cpp 检验 bo5-3.cpp 和 bo5-4.cpp 的主程序
#include"c1.h"
typedef int ElemType; // 定义矩阵元素类型 ElemType 为整型
#include"c5-3.h" // 稀疏矩阵的三元组行逻辑链接的顺序表存储结构
#include"func5-1.cpp" // comp()函数
#include"bo5-4.cpp" // 行逻辑链接稀疏矩阵存储结构的基本操作(4 个)
#define TSMatrix RLSMatrix // 将 bo5-3.cpp 和 func5-2.cpp 中的 TSMatrix 改为 RLSMatrix
#include"bo5-3.cpp" // 也可用于行逻辑链接结构三元组稀疏矩阵的基本操作(4 个)
//#define FLAG // 加此句在 func5-2.cpp 中调用 MultSMatrix1(),不加则调用 MultSMatrix()
#include"func5-2.cpp" // 主函数
```

**程序运行结果同 main5-2.cpp 的运行结果。**

## 5.3 广义表的定义

广义表,顾名思义,不是真正的线性表。它的结构更像第 6 章介绍的树结构,它的许多基本操作都是采用递归算法的。另外,还要定义如何表示一个广义表,也就是广义表的书写形式。本书采用的书写形式是字符串。算法 5.7(在 bo5-5.cpp 和 bo5-6.cpp 中)和算法 5.8(在 func5-3.cpp 中)将广义表的字符串书写形式转换为广义表的存储结构。

## 5.4 广义表的存储结构

```
// c5-4.h 广义表的头尾链表存储结构。在教科书第 109 页
enum ElemTag{ATOM,LIST};
// ATOM == 0：原子,LIST == 1：子表(见图 5-10)
typedef struct GLNode
{ ElemTag tag; // 公共部分,用于区分原子结点和表结点
  union // 原子结点和表结点的联合部分
  { AtomType atom; // atom 是原子结点的值域,AtomType 由用户定义
    struct
    { GLNode *hp, *tp;
    }ptr; // ptr 是表结点的指针域,prt.hp 和 ptr.tp 分别指向表头和表尾(表头之外的其余元素)
  };
} *GList,GLNode; // 广义表类型
```

图 5-10 广义表的头尾链表存储结构

图 5-11 是根据 c5-4.h 定义的广义表(a,(b,c,d))的存储结构。它的长度为 2,第 1 个元素为原子 a,第 2 个元素为子表(b,c,d)。

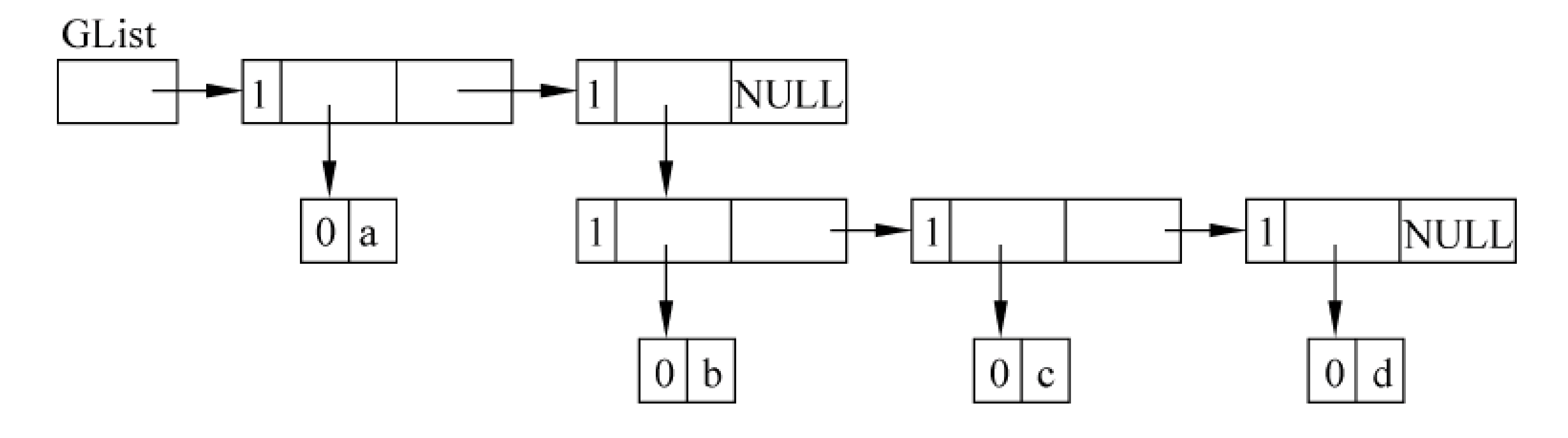



图 5-11 广义表(a,(b,c,d))的头尾链表存储结构

为了和 c5-4.h 定义的存储结构相区别,令 c5-5.h 定义的结构类型名为 GList1 和 GLNode1。

```
// c5-5.h 广义表的扩展线性链表存储结构。在教科书第 110 页(见图 5-12)
enum ElemTag{ATOM,LIST}; // ATOM == 0：原子,LIST == 1：子表
typedef struct GLNode1
{ ElemTag tag; // 公共部分,用于区分原子结点
               // 和表结点
  union // 原子结点和表结点的联合部分
  { AtomType atom; // 原子结点的值域
    GLNode1 *hp; // 表结点的表头指针
  };
  GLNode1 *tp; // 相当于线性链表的 next,指向下一个元素结点
} *GList1,GLNode1; // 广义表类型 GList1 是一种扩展的线性链表
```

图 5-12 广义表的扩展线性链表存储结构

图 5-13 是根据 c5-5.h 定义的广义表(a,(b,c,d))的扩展线性链表存储结构。在这种结构中,广义表的头指针所指结点的 tag 域值总是 1(表),其 tp 域总是 NULL。这样看来,广义表的头指针所指结点相当于表的头结点。和图 5-11 相比,图 5-13 这种结构更简洁些。

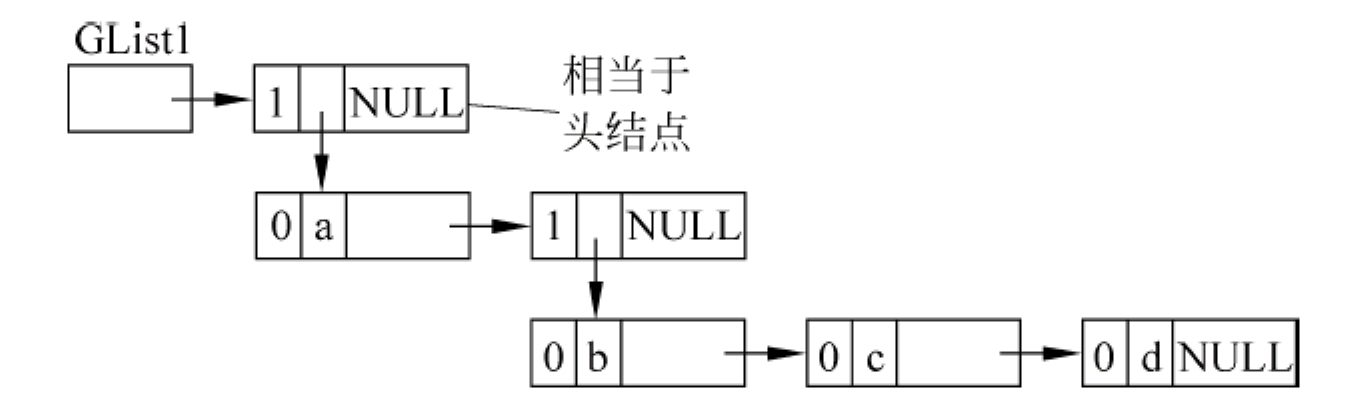



图 5-13　广义表(a,(b,c,d))的扩展线性链表存储结构

## 5.5　广义表的递归算法

```
// func5-3.cpp 广义表的书写形式串为 SString 类型,包括算法 5.8。bo5-5.cpp 和 bo5-6.cpp 调用
#include"c4-1.h" // 定义 SString 类型
#include"bo4-1.cpp" // SString 类型的基本操作
void sever(SString str,SString hstr) // 算法 5.8。SString 是数组,不需要引用类型
{ // 将脱去外层括号的非空串 str 分割成两部分:hstr 为第一个','之前的子串,str 为之后的子串
  int n,k=0,i=0; // k 记尚未配对的左括号个数
  SString ch,c1,c2,c3;
  StrAssign(c1,","); // c1=','
  StrAssign(c2,"("); // c2='('
  StrAssign(c3,")"); // c3=')'
  n=StrLength(str); // n 为串 str 的长度
  do // 搜索最外层(k=0 时)的第 1 个逗号
  {  ++i;
    SubString(ch,str,i,1); // ch 为串 str 的第 i 个字符
    if(!StrCompare(ch,c2)) // ch='('
      ++k; // 尚未配对的左括号个数+1
    else if(!StrCompare(ch,c3)) // ch=')'
      --k; // 尚未配对的左括号个数-1
  }while(i<n&&StrCompare(ch,c1)||k!=0); // i 小于串长且 ch 不是最外层的','
  if(i<n) // 串 str 中存在最外层的',',它是第 i 个字符
  { SubString(hstr,str,1,i-1); // hstr 返回串 str','前的字符
    SubString(str,str,i+1,n-i); // str 返回串 str','后的字符
  }
  else // 串 str 中不存在','
  { StrCopy(hstr,str); // 串 hstr 就是串 str
    ClearString(str); // ','后面是空串
  }
}
```

```
// bo5-5.cpp 广义表头尾链表存储结构(由 c5-4.h 定义)的基本操作(12 个),包括算法 5.5~算法 5.7
#include"func5-3.cpp" // 算法 5.8
```

```
void InitGList(GList &L)
{ // 创建空的广义表 L(见图 5-14)
  L = NULL;
}
```

NULL
L

图 5-14　空的和销毁的广义表 L

```
void CreateGList(GList &L,SString S) // 算法 5.7
{ // 采用头尾链表存储结构,由广义表的书写形式串 S 创建广义表 L。设 emp = "()"
  SString sub,hsub,emp;
  GList p,q;
  StrAssign(emp,"()"); // 空串 emp = "()"
  if(!StrCompare(S,emp)) // S = "()"
    L = NULL; // 创建空表(见图 5-14)
  else // S 不是空串
  { if(!(L = (GList)malloc(sizeof(GLNode)))) // 建表结点
      exit(OVERFLOW);
    if(StrLength(S) == 1) // S 为单原子,这种情况只会出现在递归调用中
    { L->tag = ATOM; // 创建单原子广义表
      L->atom = S[1]; // 单原子的值为字符型
    }
    else // S 为表
    { L->tag = LIST; // L 是子表
      p = L; // p 也指向子表
      SubString(sub,S,2,StrLength(S) - 2);
      // 脱外层括号(去掉第 1 个字符(左括号)和最后 1 个字符(右括号))给串 sub
      do // 重复建 n 个子表
      { sever(sub,hsub); // 从 sub 中分离出表头串给 hsub,其余部分(表尾)给 sub
        CreateGList(p->ptr.hp,hsub); // 递归创建表头串表示的子表
        q = p; // q 指向 p 所指结点
        if(!StrEmpty(sub)) // 表尾不空
        { if(!(p = (GLNode * )malloc(sizeof(GLNode)))) // 由 p 创建表结点
            exit(OVERFLOW);
          p->tag = LIST; // p 是子表
          q->ptr.tp = p; // p 所指结点接在 q 所指结点之后,形成 q 的下一个结点
        }
      }while(!StrEmpty(sub)); // 当表尾不空
      q->ptr.tp = NULL; // 设置最后一个表尾指针为空
    }
  }
}
void DestroyGList(GList &L)
{ // 销毁广义表 L(见图 5-14)
  GList q1,q2;
  if(L) // 广义表 L 不空
  { if(L->tag == LIST) // 要删除的是表结点
    { q1 = L->ptr.hp; // q1 指向表头
      q2 = L->ptr.tp; // q2 指向表尾(除表头之外的其余部分)
```

```
            DestroyGList(q1); // 递归销毁表头
            DestroyGList(q2); // 递归销毁表尾
        }
        free(L); // 释放 L 所指的存储空间(无论 L 是表结点还是原子结点)
        L = NULL; // L 不指向任何存储单元
    }
}
void CopyGList(GList &T,GList L)
{ // 采用头尾链表存储结构,由广义表 L 复制得到广义表 T。算法 5.6
  if(!L) // 复制空表
    T = NULL;
  else // 广义表 L 不空
  { T = (GList)malloc(sizeof(GLNode)); // 建表结点
    if(!T)
      exit(OVERFLOW);
    T->tag = L->tag; // 复制标志域
    if(L->tag == ATOM) // 单原子
      T->atom = L->atom; // 复制单原子
    else // 子表
    { CopyGList(T->ptr.hp,L->ptr.hp); // 递归复制表头子表
      CopyGList(T->ptr.tp,L->ptr.tp); // 递归复制表尾(除表头之外的部分)子表
    }
  }
}
int GListLength(GList L)
{ // 返回广义表的长度,即元素个数
  int len = 0; // 设置广义表长度的初值为 0
  while(L) // 未到表尾
  { L = L->ptr.tp; // L 指向广义表最外层的下一个元素
    len ++; // 表长 + 1
  }
  return len;
}
int GListDepth(GList L)
{ // 采用头尾链表存储结构,求广义表 L 的深度。算法 5.5
  int dep,max = 0;
  GList pp;
  if(!L) // 广义表 L 为空
    return 1; // 空表深度为 1
  if(L->tag == ATOM) // 是原子结点
    return 0; // 原子深度为 0,只会出现在递归调用中
  for(pp = L;pp;pp = pp->ptr.tp) // 从本层的第 1 个元素到最后一个元素
  { dep = GListDepth(pp->ptr.hp); // 递归求以 pp->ptr.hp 为头指针的子表深度
    if(dep>max)
      max = dep; // max 存本层子表深度的最大值
```

```
  }
  return max + 1; // 非空表的深度是各元素的深度的最大值加 1
}
Status GListEmpty(GList L)
{ // 判定广义表是否为空
  if(!L)
    return TRUE;
  else
    return FALSE;
}
GList GetHead(GList L)
{ // 生成广义表 L 的表头元素,返回指向这个元素的指针
  GList h;
  if(!L) // 空表无表头
    return NULL;
  CopyGList(h,L->ptr.hp); // 将 L 的表头元素复制给 h
  return h;
}
GList GetTail(GList L)
{ // 将广义表 L 的表尾(除表头之外的部分)生成为广义表,返回指向这个新广义表的指针
  GList t;
  if(!L) // 空表无表尾
    return NULL;
  CopyGList(t,L->ptr.tp); // 将 L 的表尾元素复制给 t
  return t;
}
void InsertFirst_GL(GList &L,GList e)
{ // 初始条件:广义表 L 存在。操作结果:插入元素 e(也可能是子表)作为广义表 L 的第 1 个元素(表头)
  GList p = (GList)malloc(sizeof(GLNode)); // 生成新的表头结点
  if(!p)
    exit(OVERFLOW);
  p->tag = LIST; // 广义表 L 的类型是表
  p->ptr.hp = e; // L 的表头指针指向 e
  p->ptr.tp = L; // L 的表尾指针指向原表 L
  L = p; // L 指向新的表头结点
}
void DeleteFirst_GL(GList &L,GList &e)
{ // 初始条件:广义表 L 存在。操作结果:删除广义表 L 的第 1 个元素(表头),并用 e 返回其值
  GList p = L; // p 指向第 1 个表结点
  e = L->ptr.hp; // e 指向 L 的表头元素
  L = L->ptr.tp; // L 指向原 L 的表尾(除表头之外的部分)
  free(p); // 释放 p 所指的表结点
}
void Traverse_GL(GList L,void(* visit)(AtomType))
{ // 利用递归算法遍历广义表 L
```

```
  if(L) // L不空
    if(L->tag == ATOM) // L为单原子
      visit(L->atom);
    else // L为广义表
    { Traverse_GL(L->ptr.hp,visit); // 递归遍历L的表头
      Traverse_GL(L->ptr.tp,visit); // 递归遍历L的表尾
    }
}
```

```
// func5-4.cpp 广义表的主函数,main5-4.cpp和main5-5.cpp调用
void main()
{
  char p[80];
  SString t;
  GList n,m;
  InitGList(n); // 初始化广义表n,n为空表
  printf("空广义表n的深度=%d,n是否空?%d(1:是 0:否)\n",GListDepth(n),
  GListEmpty(n));
  printf("请输入广义表n(书写形式:空表:(),单原子:(a),其他:(a,(b),c)):\n");
  gets(p); // 将描述广义表n的字符串赋给p
  StrAssign(t,p); // 将p转换为SString类型的t
  CreateGList(n,t); // 根据t创建广义表n
  printf("广义表n的长度=%d,",GListLength(n));
  printf("深度=%d,n是否空?%d(1:是 0:否)\n",GListDepth(n),GListEmpty(n));
  printf("遍历广义表n:");
  Traverse_GL(n,print2); // 遍历广义表n
  CopyGList(m,n); // 复制广义表m=n
  printf("\n复制广义表m=n,m的长度=%d,",GListLength(m));
  printf("m的深度=%d\n遍历广义表m:",GListDepth(m));
  Traverse_GL(m,print2); // 遍历广义表m
  DestroyGList(m); // 销毁广义表m,释放存储空间
  m = GetHead(n); // 生成广义表n的表头元素,并由m指向
  printf("\nm是n的表头元素,遍历m:");
  Traverse_GL(m,print2); // 遍历广义表m
  DestroyGList(m); // 销毁广义表m,释放存储空间
  m = GetTail(n); // 将广义表n的表尾(除表头之外的部分)生成为广义表,并由m指向其
  printf("\nm是由n的表尾形成的广义表,遍历广义表m:");
  Traverse_GL(m,print2); // 遍历广义表m
  InsertFirst_GL(m,n); // 将广义表n插到广义表m中,并作为m的第1个元素(表头)
  printf("\n插入广义表n为m的表头,遍历广义表m:");
  Traverse_GL(m,print2); // 遍历广义表m
  DeleteFirst_GL(m,n); // 删除广义表m的表头,并由n指向删除的表头
  printf("\n删除m的表头,并由n指向m的表头,遍历广义表m:");
  Traverse_GL(m,print2); // 遍历广义表m
```

```
  printf("\n遍历广义表n(广义表m的原表头):");
  Traverse_GL(n,print2); // 遍历广义表n
  printf("\n");
  DestroyGList(m); // 销毁广义表m、n,并释放存储空间
  DestroyGList(n);
}
```

```
// main5-4.cpp 检验bo5-5.cpp的主程序
#include"c1.h"
typedef char AtomType; // 定义原子类型为字符型
#include"c5-4.h" // 定义广义表的头尾链表存储
#include"bo5-5.cpp" // 广义表的头尾链表存储结构的基本操作(12个)
#define ElemType AtomType // 将func2-2.cpp中的ElemType类型定义为AtomType类型
#include"func2-2.cpp" // 包括equal()、comp()、print()、print1()和print2()函数
#include"func5-4.cpp" // 主函数
```

**程序运行结果**(以教科书图5.7为例):

```
空广义表n的深度=1,n是否空? 1(1:是 0:否)
请输入广义表n(书写形式:空表:(),单原子:(a),其他:(a,(b),c)):
((),(e),(a,(b,c,d)))↙(见图5-15)
广义表n的长度=3,深度=3,n是否空? 0(1:是 0:否)
遍历广义表n:e a b c d
复制广义表m=n,m的长度=3,m的深度=3
遍历广义表m:e a b c d
m是n的表头元素,遍历m:(见图5-16)
m是由n的表尾形成的广义表,遍历广义表m:e a b c d(见图5-17)
插入广义表n为m的表头,遍历广义表m:e a b c d e a b c d(见图5-18)
删除m的表头,并由n指向m的表头,遍历广义表m:e a b c d(见图5-17)
遍历广义表n(广义表m的原表头):e a b c d(见图5-15)
```

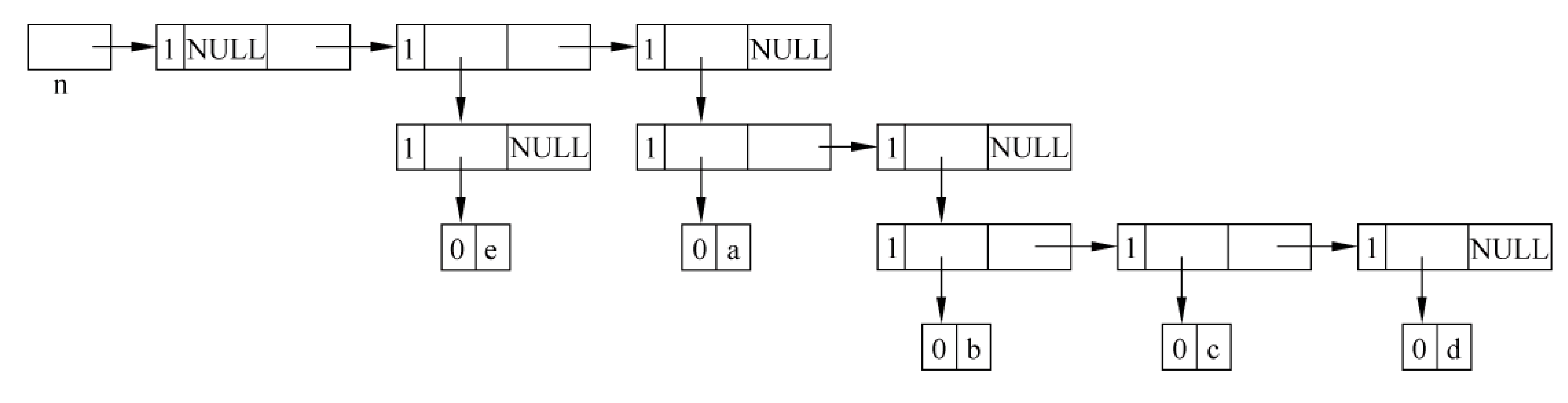



图5-15　广义表((),(e),(a,(b,c,d)))的头尾链表存储结构

m
NULL

图5-16　广义表n的表头元素m

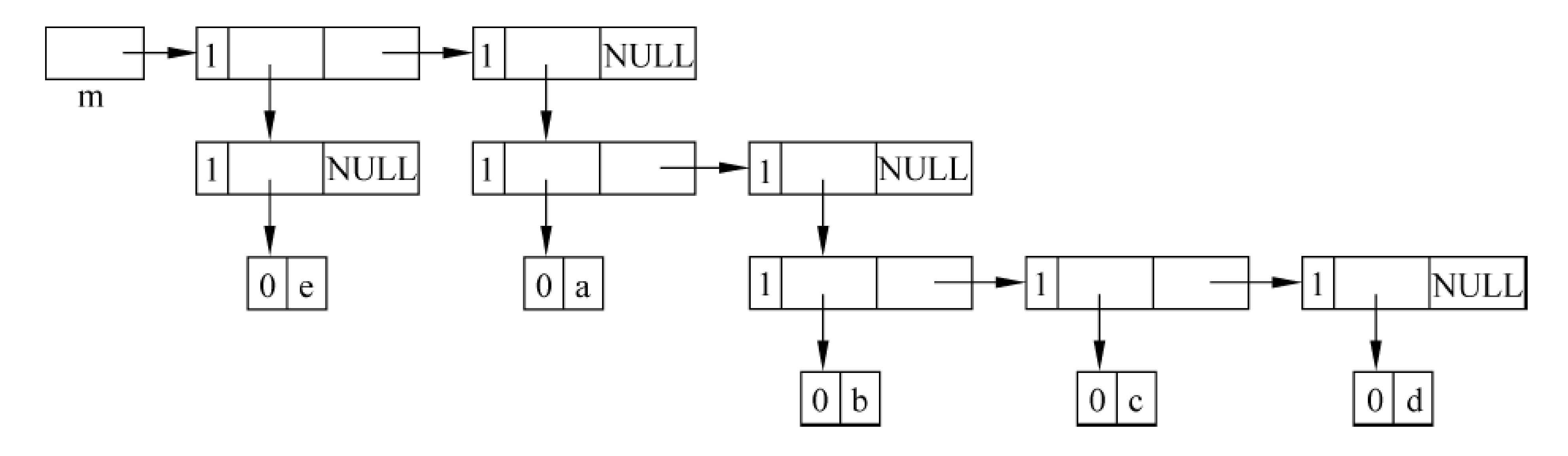



图 5-17　由广义表 n 的表尾形成的广义表 m

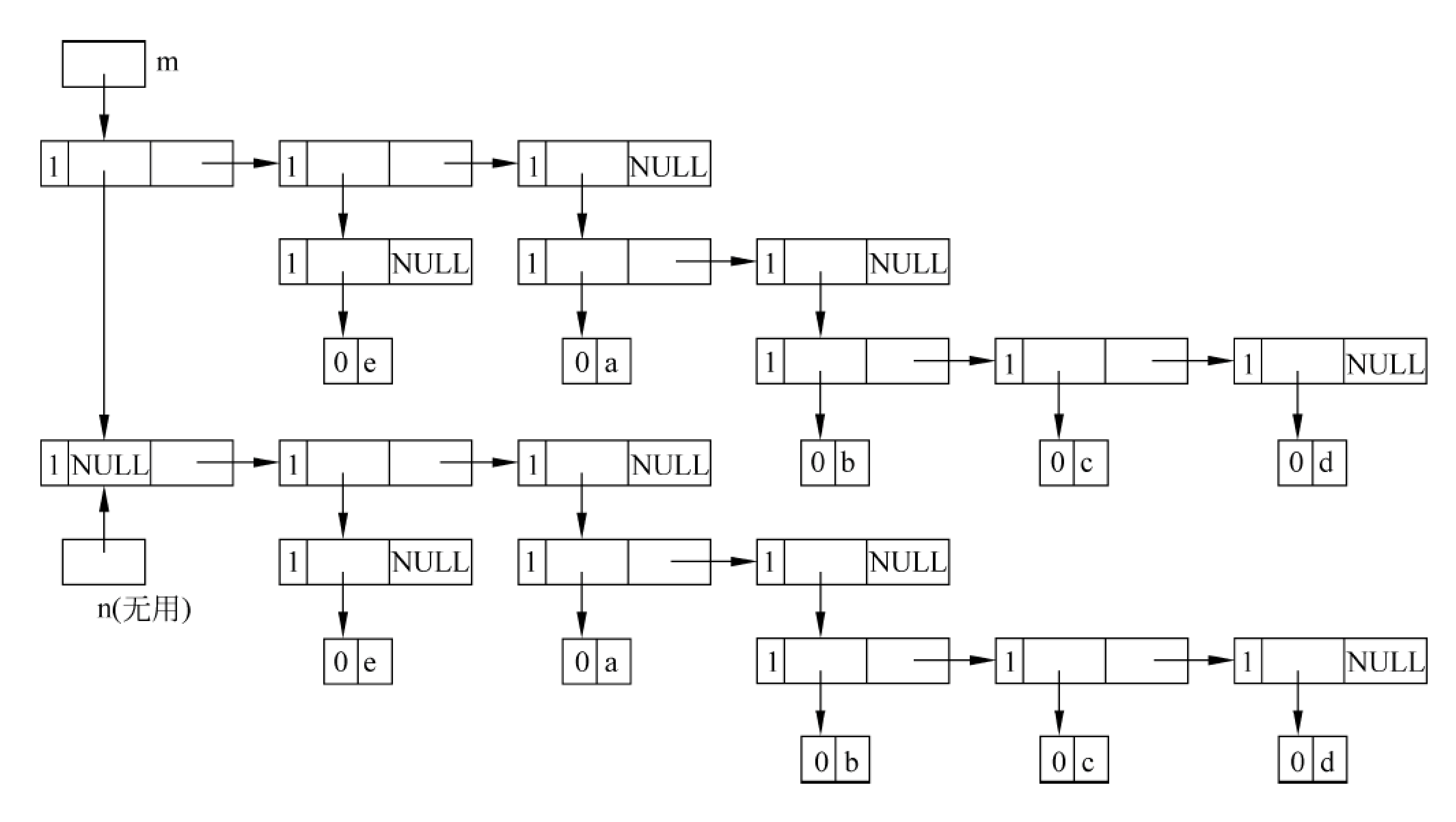



图 5-18　将广义表 n 插在广义表 m 中并作为广义表 m 的表头

```
// bo5-6.cpp 广义表的扩展线性链表存储(存储结构由 c5-5.h 定义)的基本操作(12 个)
#include"func5-3.cpp" // 算法 5.8
void InitGList(GList1 &L)
{ // 创建空的广义表 L(见图 5-19)
  L = (GList1)malloc(sizeof(GLNode1)); // 建表头结点
  L->tag = LIST; // 给表头结点的标志域赋值
  L->hp = L->tp = NULL; // 给表头结点的指针域赋值
}
```

L　1 NULL NULL

图 5-19　空的广义表 L

```
void Create(GList1 &L,SString S)
{ // 由广义表的书写形式串 S 创建子表 L,CreateGList()调用
  SString emp,sub,hsub;
  GList1 p;
  StrAssign(emp,"()"); // 设 emp = "()"
  if(!(L = (GList1)malloc(sizeof(GLNode1)))) // 建表结点不成功
    exit(OVERFLOW);
  if(!StrCompare(S,emp)) // 创建空表
  { L->tag = LIST; // 依然是表
    L->hp = NULL; // 是空表
```

```
    }
    else if(StrLength(S) == 1) // 创建单原子广义表
    { L->tag = ATOM; // 给结点的标志域赋值
      L->atom = S[1]; // 给结点的值域赋值
    }
    else // 递归创建子表
    { L->tag = LIST; // 是表
      SubString(sub,S,2,StrLength(S) - 2);
      // 脱外层括号(去掉第 1 个字符(左括号)和最后 1 个字符(右括号))给串 sub
      sever(sub,hsub); // 从 sub 中分离出表头串 hsub(外层逗号之前的),表尾串赋给 sub
      Create(L->hp,hsub); // 创建子表的表头元素
      p = L->hp; // p 指向子表的表头元素
      while(!StrEmpty(sub)) // 表尾不空,则重复建 n 个子表
      { sever(sub,hsub); // 从 sub 中分离出表头串 hsub
        Create(p->tp,hsub); // 依次创建子表
        p = p->tp; // p 向后移
      }
    }
    L->tp = NULL; // 尾指针为空
}
void CreateGList(GList1 &L,SString S) // 修改算法 5.7
{ // 采用扩展线性链表存储结构,由广义表的书写形式串 S 创建广义表 L。L 最初为空的广义表
    SString emp,sub,hsub;
    GList1 p;
    StrAssign(emp,"()"); // 设 emp = "()"
    if(! StrCompare(S,emp)) // S = "()"
      InitGList(L); // 创建空表
    else // S 不是空串
    { SubString(sub,S,2,StrLength(S) - 2);
      // 脱外层括号(去掉第 1 个字符(左括号)和最后一个字符(右括号))给串 sub
      sever(sub,hsub); // 从 sub 中分离出表头串 hsub(外层逗号之前的),表尾串赋给 sub
      Create(L->hp,hsub); // 创建表头元素
      p = L->hp; // p 指向第 1 个元素
      while(!StrEmpty(sub)) // 表尾不空,则继续创建子表
      { sever(sub,hsub); // 从 sub 中分离出最前面的串给 hsub,其余部分赋给 sub
        Create(p->tp,hsub); // 依次创建子表
        p = p->tp; // p 向后移
      }
      p->tp = NULL; // 最后一个元素的尾指针为空
    }
}
void DestroyGList(GList1 &L)
{ // 初始条件:广义表 L 存在。操作结果:销毁广义表 L(见图 5-20)
    GList1 ph,pt;
    if(L) // L 存在
```

| NULL |
|---|

L

图 5-20　销毁的广义表 L

```
  { // 由 ph 和 pt 接替 L 的两个指针
    if(L->tag) // 是子表
      ph = L->hp;
    else // 是原子
      ph = NULL;
    pt = L->tp;
    DestroyGList(ph); // 递归销毁表 ph
    DestroyGList(pt); // 递归销毁表 pt
    free(L); // 释放 L 所指结点
    L = NULL; // 令 L 为空
  }
}
void CopyGList(GList1 &T,GList1 L)
{ // 初始条件:广义表 L 存在。操作结果:由广义表 L 复制得到广义表 T
  T = NULL;
  if(L) // L 存在
  { T = (GList1)malloc(sizeof(GLNode1));
    if(!T)
      exit(OVERFLOW);
    T->tag = L->tag; // 复制枚举变量
    if(L->tag == ATOM) // 复制共用体部分
      T->atom = L->atom; // 复制单原子
    else
      CopyGList(T->hp,L->hp); // 复制子表
    if(L->tp == NULL) // 到表尾
      T->tp = L->tp;
    else
      CopyGList(T->tp,L->tp); // 复制子表
  }
}
int GListLength(GList1 L)
{ // 初始条件:广义表 L 存在。操作结果:求广义表 L 的长度,即元素个数
  int len = 0;
  GList1 p = L->hp; // p 指向第 1 个元素
  while(p)
  { len ++;
    p = p->tp;
  };
  return len;
}
int GListDepth(GList1 L)
{ // 初始条件:广义表 L 存在。操作结果:求广义表 L 的深度
  int max = 0,dep;
  GList1 p;
  if(L->tag == LIST&&!L->hp)
```

```
    return 1; // 空表深度为 1
  else if(L->tag == ATOM)
    return 0; // 单原子表深度为 0,只会出现在递归调用中
  else // 求一般表的深度
    for(p = L->hp;p;p = p->tp)
    { dep = GListDepth(p); // 求以 p 为头指针的子表深度
      if(dep>max)
        max = dep;
    }
  return max + 1; // 非空表的深度是各元素的深度的最大值加 1
}
Status GListEmpty(GList1 L)
{ // 初始条件:广义表 L 存在。操作结果:判定广义表 L 是否为空
  if(!L->hp)
    return OK;
  else
    return ERROR;
}
GList1 GetHead(GList1 L)
{ // 初始条件:广义表 L 存在。操作结果:生成广义表 L 的表头元素,返回指向这个元素的指针
  GList1 h,p;
  if(!L->hp) // 空表无表头
    return NULL;
  p = L->hp->tp; // p 指向 L 的表尾
  L->hp->tp = NULL; // 截去 L 的表尾部分
  CopyGList(h,L->hp); // 将表头元素复制给 h
  L->hp->tp = p; // 恢复 L 的表尾(保持原 L 不变)
  return h;
}
GList1 GetTail(GList1 L)
{ // 初始条件:广义表 L 存在。操作结果:将 L 的表尾生成为广义表,返回指向这个新广义表的指针
  GList1 t;
  InitGList(t); // 创建空的广义表 t
  if(L->hp) // L 非空
    CopyGList(t->hp,L->hp->tp); // 将 L 的表尾复制给 t 的表头
  return t;
}
void InsertFirst_GL(GList1 &L,GList1 e)
{ // 初始条件:广义表 L 存在。操作结果:插入元素 e(也可能是子表)作为 L 的第 1 个元素(表头)
  GList1 p = L->hp; // p 指向广义表 L 的第 1 个元素
  L->hp = e; // 广义表 L 的头指针指向 e
  e->tp = p; // e(只是 1 个元素,其尾指针必为 NULL)的尾指针指向 L 原来的第 1 个元素
}
void DeleteFirst_GL(GList1 &L,GList1 &e)
```

```
{ // 初始条件：广义表 L 存在。操作结果：删除广义表 L 的第 1 个元素，并用 e 返回其指针
  e = L->hp; // e 指向 L 的第 1 个元素
  if(L->hp) // L 非空
  { L->hp = e->tp; // L 的头指针指向原第 2 个元素
    e->tp = NULL; // e 的尾指针设为空
  }
}
void Traverse_GL(GList1 L,void(* visit)(AtomType))
{ // 利用递归算法遍历广义表 L
  if(L) // L 存在
  { if(L->tag == ATOM) // L 为单原子
      visit(L->atom); // 访问单原子
    else // L 为子表
      Traverse_GL(L->hp,visit); // 遍历 L->hp 子表
    Traverse_GL(L->tp,visit); // 遍历 L->tp 子表
  }
}
```

```
// main5-5.cpp 检验 bo5-6.cpp 的主程序
#include"c1.h"
typedef char AtomType; // 定义原子类型为字符型
#include"c5-5.h" // 定义广义表的扩展线性链表存储结构
#include"bo5-6.cpp" // 广义表的扩展线性链表存储结构基本操作
#define ElemType AtomType // 将 func2-2.cpp 中的 ElemType 类型定义为 AtomType 类型
#include"func2-2.cpp" // 包括 equal()、comp()、print()、print1()和 print2()函数
#define GList GList1 // 定义 func5-4.cpp 中的 GList 类型为 GList1 类型
#include"func5-4.cpp" // 主函数
```

**程序运行结果：**

```
空广义表 n 的深度 = 1,n 是否空？1(1:是 0:否)
请输入广义表 n(书写形式：空表:(),单原子:(a),其他:(a,(b),c))：
(a,(b),c)↙(见图 5-21)
广义表 n 的长度=3,深度=2,n 是否空？0(1:是 0:否)
遍历广义表 n：a b c
复制广义表 m=n,m 的长度=3,m 的深度=2
遍历广义表 m：a b c
m 是 n 的表头元素,遍历 m：a (见图 5-22)
m 是由 n 的表尾形成的广义表,遍历广义表 m：b c (见图 5-23)
插入广义表 n 为 m 的表头,遍历广义表 m：a b c b c (见图 5-24)
删除 m 的表头,并由 n 指向 m 的表头,遍历广义表 m：b c (见图 5-23)
遍历广义表 n(广义表 m 的原表头)：a b c (见图 5-21)
```

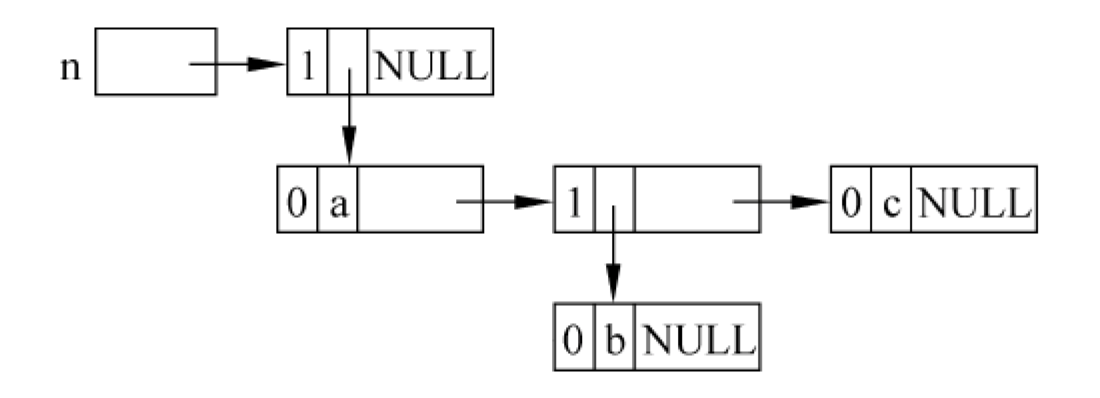



图 5-21　广义表(a,(b),c)的扩展线性链表存储结构

图 5-22　广义表 n 的表头元素 m

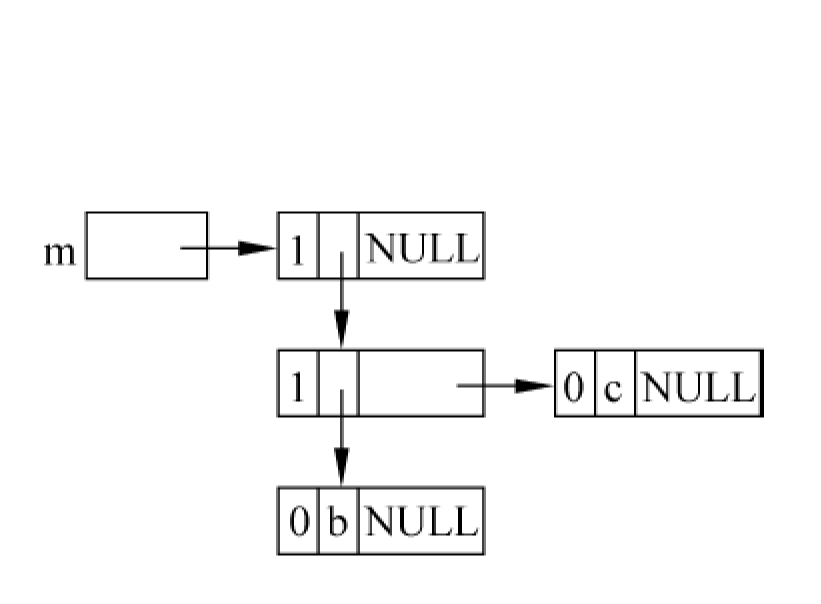



图 5-23　由广义表 n 的表尾形成的广义表 m

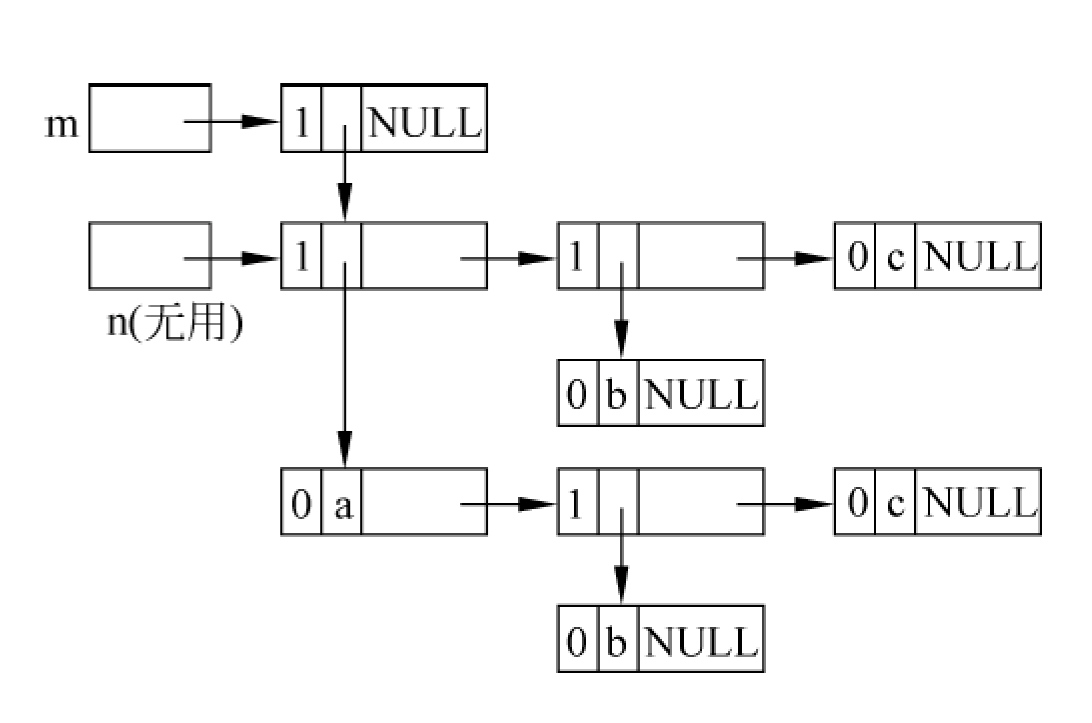



图 5-24　将广义表 n 插在广义表 m 中并作为广义表 m 的表头

# 第 6 章 树和二叉树

## 6.1 二　叉　树

```
// c6-1.h 二叉树的顺序存储结构。在教科书第 126 页(见图 6-1)
#define MAX_TREE_SIZE 100 // 二叉树的最大结点数
typedef TElemType SqBiTree[MAX_TREE_SIZE]; // 0 号单元存储根结点
struct position // 新增
{ int level,order; // 结点所在的层,在该层的序号(按满二叉树计算)
};
```

在顺序存储结构中,二叉树的每一个结点,根据它所在的层、在该层的排序(按满二叉树计),都有一个固定的序号。如图 6-2 所示,根结点的序号为 0;第 i 层结点的序号从 $2^{i-1}-1$～$2^i-2$;序号为 j 的结点,其所处的层 i 是$\lceil \log_2(j+2) \rceil$;其在层 i 中的排序 $k=j+2-2^{i-1}$;其双亲序号为$\lfloor (j-1)/2 \rfloor$;其左右孩子序号分别为 2j+1 和 2j+2。除了根结点,序号为奇数的结点是其双亲的左孩子,它的右兄弟的序号是它的序号+1;序号为偶数的结点是其双亲的右孩子,它的左兄弟的序号是它的序号−1。以序号为 5 的结点为例,它处于第 3 层,而 3 等于$\lceil \log_2(5+2) \rceil$;它在第 3 层中的排序是 3;其双亲序号为 2;其左右孩子序号分别为 11 和 12;它是其双亲的左孩子;它的右兄弟的序号是 6。i 层的满二叉树,其结点总数为 $2^i-1$。

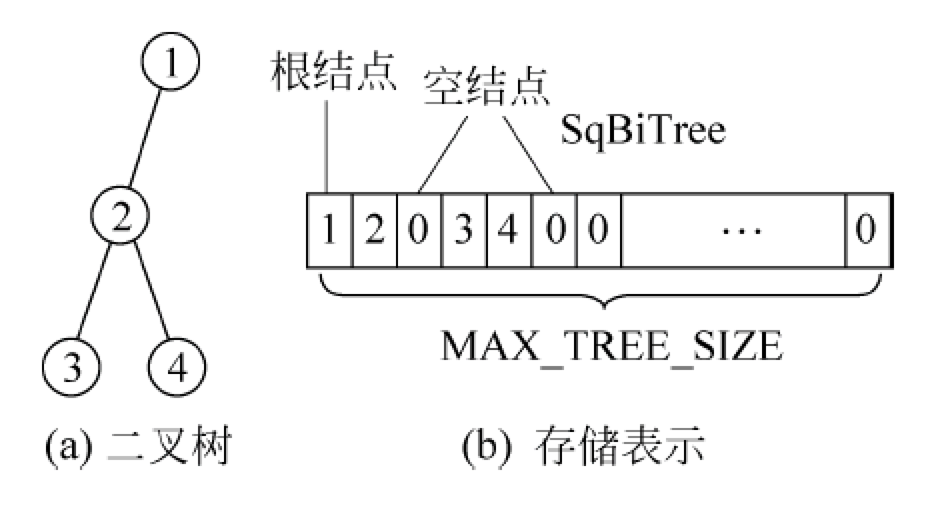



图 6-1　二叉树的顺序存储结构

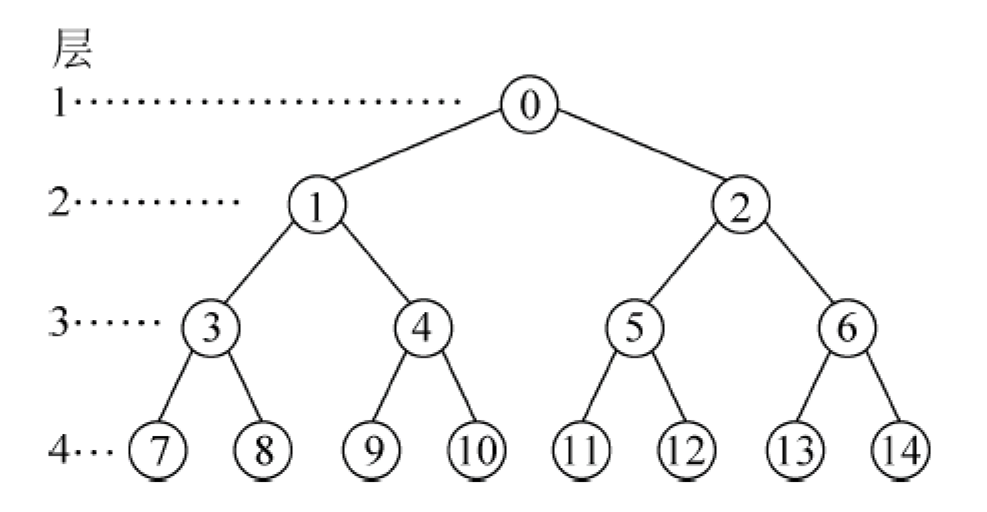



图 6-2　顺序存储结构的二叉树序号

注:圈内数字为该结点在数组中的序号

显然,在顺序存储结构中,按层序输入二叉树是最方便的。当最后一个结点的值输入后,输入给定符号表示结束。二叉树的顺序存储结构适合存完全二叉树或近似完全二叉树。第 10 章的树形选择排序和堆排序,以及第 11 章的多路平衡归并和置换-选择排序都要建立

完全二叉树，故采用顺序存储结构。

bo6-1.cpp 是采用顺序存储结构的基本操作函数，main6-1.cpp 是检验这些基本操作的主程序。main6-1.cpp、main6-2.cpp 和 main6-3.cpp 采用了编译预处理的"#define CHAR 1"、"#if CHAR"等命令。这样，只要将这些程序的第 2 行和第 3 行其中之一作为注释行，程序就可以分别在结点类型为整型或字符型的情况下应用而不用做其他改动。

```
// bo6-1.cpp 二叉树的顺序存储(存储结构由 c6-1.h 定义)的基本操作(23 个)
#define ClearBiTree InitBiTree // 在顺序存储结构中，两函数完全一样
#define DestroyBiTree InitBiTree // 在顺序存储结构中，两函数完全一样
void InitBiTree(SqBiTree T)
{ // 构造空二叉树 T。因为 T 是数组名，故不需要 &
  int i;
  for(i=0;i<MAX_TREE_SIZE;i++)
    T[i]=Nil; // 初值为空(Nil 在主程中定义)
}
void CreateBiTree(SqBiTree T)
{ // 按层序次序输入二叉树中结点的值(字符型或整型)，构造顺序存储的二叉树 T
  int i=0;
  InitBiTree(T); // 构造空二叉树 T
#if CHAR // CHAR 值为非零，结点类型为字符
  int n;
  char s[MAX_TREE_SIZE];
  printf("请按层序输入结点的值(字符)，空格表示空结点，结点数≤%d:\n",MAX_TREE_SIZE);
  gets(s); // 输入字符串
  n=strlen(s); // 求字符串的长度
  for(;i<n;i++) // 将字符串赋值给 T
    T[i]=s[i];
#else // CHAR 值为零，结点类型为整型
  printf("请按层序输入结点的值(整型)，0 表示空结点，输 999 结束。结点数≤%d:\n",
  MAX_TREE_SIZE);
  while(1) // 永真循环
  { scanf("%d",&T[i]); // 按层序依次输入
    if(T[i]==999) // 输入结束
    { T[i]=Nil; // 恢复为空结点
      break; // 跳出循环
    }
    i++; // 计数加 1
  }
#endif
  for(i=1;i<MAX_TREE_SIZE;i++) // 从第 2 个(非根)结点开始检查
    if(T[i]!=Nil&&T[(i+1)/2-1]==Nil) // 此结点不空但无双亲
    { printf("出现无双亲的非根结点"form"\n",T[i]);
      exit(OVERFLOW);
    }
}
```

```
Status BiTreeEmpty(SqBiTree T)
{ // 初始条件：二叉树 T 存在。操作结果：若 T 为空二叉树，则返回 TRUE；否则 FALSE
  if(T[0] == Nil) // 根结点为空，则树空
    return TRUE;
  else
    return FALSE;
}
int BiTreeDepth(SqBiTree T)
{ // 初始条件：二叉树 T 存在。操作结果：返回 T 的深度
  int i;
  if(T[0] == Nil) // 根结点为空，则树空
    return 0; // 空树的深度为 0
  for(i = MAX_TREE_SIZE - 1;i>= 0;i-- ) // 从数组的后面向前找起
    if(T[i]!= Nil) // 找到最后一个结点，其序号为 i
      break;
  return (int)(log(i + 1)/log(2) + 1.1); // 序号为 i 的结点的深度就是树的深度
}
Status Root(SqBiTree T,TElemType &e)
{ // 初始条件：二叉树 T 存在
  // 操作结果：当 T 不空，用 e 返回 T 的根，返回 OK；否则返回 ERROR，e 无定义
  if(BiTreeEmpty(T)) // T 空
    return ERROR;
  else
  { e = T[0];
    return OK;
  }
}
TElemType Value(SqBiTree T,position e)
{ // 初始条件：二叉树 T 存在，e 是 T 中某个结点(的位置)
  // 操作结果：返回处于位置 e(层，本层序号)的结点的值
  return T[int(pow(2,e.level - 1) + e.order - 2)];
}
Status Assign(SqBiTree T,position e,TElemType value)
{ // 初始条件：二叉树 T 存在，e 是 T 中某个结点(的位置)
  // 操作结果：给处于位置 e(层，本层序号)的结点赋新值 value
  int i = int(pow(2,e.level - 1) + e.order - 2); // 将层、本层序号转为数组的序号
  if(i!= 0&&value!= Nil&&T[(i + 1)/2 - 1] == Nil) // 不是根结点，值非空，但双亲为空
    return ERROR;
  else if(value == Nil&&(T[i * 2 + 1]!= Nil||T[i * 2 + 2]!= Nil)) // 给双亲赋空值但有孩子结点
    return ERROR;
  T[i] = value; // 以上 2 种情况之外，给结点赋新值
  return OK;
}
TElemType Parent(SqBiTree T,TElemType e)
{ // 初始条件：二叉树 T 存在，e 是 T 中某个结点
```

```
  // 操作结果：若 e 是 T 的非根结点，则返回它的双亲；否则返回“空”
  int i;
  if(T[0] == Nil) // 空树
    return Nil; // 返回“空”
  for(i = 1;i<= MAX_TREE_SIZE - 1;i++ ) // 从二叉树的第 2 个结点开始查找
    if(T[i] == e) // 找到 e
      return T[(i + 1)/2 - 1]; // 返回其双亲结点的值
  return Nil; // 未找到 e
}
TElemType LeftChild(SqBiTree T,TElemType e)
{ // 初始条件：二叉树 T 存在，e 是 T 中某个结点
  // 操作结果：返回 e 的左孩子。若 e 无左孩子，则返回“空”
  int i;
  for(i = 0;i<= (MAX_TREE_SIZE - 2)/2;i++ ) // 从 T 的第 1 个结点到最后一个可能有左孩子的结点
    if(T[i] == e) // 找到 e
      return T[i * 2 + 1]; // 返回 e 的左孩子的值
  return Nil; // 未找到 e
}
TElemType RightChild(SqBiTree T,TElemType e)
{ // 初始条件：二叉树 T 存在，e 是 T 中某个结点
  // 操作结果：返回 e 的右孩子。若 e 无右孩子，则返回“空”
  int i;
  for(i = 0;i<= (MAX_TREE_SIZE - 3)/2;i++ ) // 从 T 的第 1 个结点到最后一个可能有右孩子的结点
    if(T[i] == e) // 找到 e
      return T[i * 2 + 2]; // 返回 e 的右孩子的值
  return Nil; // 未找到 e
}
TElemType LeftSibling(SqBiTree T,TElemType e)
{ // 初始条件：二叉树 T 存在，e 是 T 中某个结点
  // 操作结果：返回 e 的左兄弟。若 e 是 T 的左孩子或无左兄弟，则返回“空”
  int i;
  if(T[0] == Nil) // 空树
    return Nil; // 返回“空”
  for(i = 1;i<= MAX_TREE_SIZE - 1;i++ ) // 从二叉树 T 的第 2 个结点开始查找
    if(T[i] == e&&i %2 == 0) // 找到 e 且其序号为偶数(是右孩子)
      return T[i - 1]; // 返回 e 的左兄弟的值
  return Nil; // 未找到 e
}
TElemType RightSibling(SqBiTree T,TElemType e)
{ // 初始条件：二叉树 T 存在，e 是 T 中某个结点
  // 操作结果：返回 e 的右兄弟。若 e 是 T 的右孩子或无右兄弟，则返回“空”
  int i;
  if(T[0] == Nil) // 空树
    return Nil; // 返回“空”
  for(i = 1;i<= MAX_TREE_SIZE - 2;i++ ) // 从二叉树 T 的第 2 个结点开始查找
```

```
    if(T[i] == e&&i%2) // 找到e且其序号为奇数(是左孩子)
      return T[i+1]; // 返回e的右兄弟的值
  return Nil; // 未找到e
}
void Move(SqBiTree q,int j,SqBiTree T,int i) // InsertChild()用到。新增
{ // 把从q的j结点开始的子树移为从T的i结点开始的子树
  if(i>=MAX_TREE_SIZE) // i结点超出了存储范围
    exit(OVERFLOW);
  if(q[2*j+1]!=Nil) // q的左子树不空
    Move(q,(2*j+1),T,(2*i+1)); // 把q的j结点的左子树移为T的i结点的左子树
  if(q[2*j+2]!=Nil) // q的右子树不空
    Move(q,(2*j+2),T,(2*i+2)); // 把q的j结点的右子树移为T的i结点的右子树
  T[i]=q[j]; // 把q的j结点移为T的i结点
  q[j]=Nil; // 把q的j结点置空
}
void InsertChild(SqBiTree T,TElemType p,int LR,SqBiTree c)
{ // 初始条件:二叉树T存在,p是T中某个结点的值,LR为0或1,
  //           非空二叉树c与T不相交且右子树为空
  // 操作结果:根据LR为0或1,插入c为T中p结点的左或右子树。
  //           p结点的原有左或右子树则成为c的右子树
  int j,k;
  for(j=0;j<int(pow(2,BiTreeDepth(T)))-1;j++) // 查找p的序号
    if(T[j]==p) // j为p的序号
      break;
  k=2*j+1+LR; // k为p的左或右孩子的序号
  if(T[k]!=Nil) // p原来的左或右孩子不空
    Move(T,k,c,2); // 把从T的k结点开始的子树移为c的右子树
  Move(c,0,T,k); // 把树c移为从T的k结点开始的子树
}
typedef int QElemType; // 定义队列元素类型为整型(序号)
#include"c3-2.h" // 链队列
#include"bo3-2.cpp" // 链队列的基本操作
Status DeleteChild(SqBiTree T,position p,int LR)
{ // 初始条件:二叉树T存在,p指向T中某个结点,LR为0或1
  // 操作结果:根据LR为0或1,删除T中p所指结点的左或右子树
  int i;
  Status k=OK; // 队列不空的标志
  LinkQueue q;
  InitQueue(q); // 初始化队列,用于存放待删除的结点
  i=(int)pow(2,p.level-1)+p.order-2; // 将层、本层序号转为数组的序号
  if(T[i]==Nil) // 此结点空
    return ERROR;
  i=i*2+1+LR; // 待删除子树的根结点在数组中的序号
  while(k)
  { if(T[2*i+1]!=Nil) // 左结点不空
```

```
      EnQueue(q,2 * i + 1); // 入队左结点的序号
    if(T[2 * i + 2]!= Nil) // 右结点不空
      EnQueue(q,2 * i + 2); // 入队右结点的序号
    T[i] = Nil; // 删除此结点
    k = DeQueue(q,i); // 出队结点的序号,其值赋给 i,成功(队列不空)返回 OK;否则返回 ERROR
  }
  return OK;
}
void( * VisitFunc)(TElemType); // 函数变量
void PreTraverse(SqBiTree T,int e)
{ // 递归先序遍历二叉树 T 中序号为 e 的子树,PreOrderTraverse()调用
  VisitFunc(T[e]); // 访问树 T 中序号为 e 的结点
  if(T[2 * e + 1]!= Nil) // 序号为 e 的结点的左子树不空
    PreTraverse(T,2 * e + 1); // 递归先序遍历树 T 中序号为 e 的结点的左子树
  if(T[2 * e + 2]!= Nil) // 序号为 e 的结点的右子树不空
    PreTraverse(T,2 * e + 2); // 递归先序遍历树 T 中序号为 e 的结点的右子树
}
void PreOrderTraverse(SqBiTree T,void(* Visit)(TElemType))
{ // 初始条件:二叉树存在,Visit 是对结点操作的应用函数
  // 操作结果:先序遍历 T,对每个结点调用函数 Visit 一次且仅一次
  VisitFunc = Visit;
  if(!BiTreeEmpty(T)) // 树不空
    PreTraverse(T,0); // 递归先序遍历树 T 中序号为 0 的树(树 T 自身)
  printf("\n");
}
void InTraverse(SqBiTree T,int e)
{ // 递归中序遍历二叉树 T 中序号为 e 的子树,InOrderTraverse()调用
  if(T[2 * e + 1]!= Nil) // 序号为 e 的结点的左子树不空
    InTraverse(T,2 * e + 1); // 递归中序遍历树 T 中序号为 e 的结点的左子树
  VisitFunc(T[e]); // 访问树 T 中序号为 e 的结点
  if(T[2 * e + 2]!= Nil) // 序号为 e 的结点的右子树不空
    InTraverse(T,2 * e + 2); // 递归中序遍历树 T 中序号为 e 的结点的右子树
}
void InOrderTraverse(SqBiTree T,void(* Visit)(TElemType))
{ // 初始条件:二叉树存在,Visit 是对结点操作的应用函数
  // 操作结果:中序遍历 T,对每个结点调用函数 Visit 一次且仅一次
  VisitFunc = Visit;
  if(!BiTreeEmpty(T)) // 树不空
    InTraverse(T,0); // 递归中序遍历树 T 中序号为 0 的树(树 T 自身)
  printf("\n");
}
void PostTraverse(SqBiTree T,int e)
{ // 递归后序遍历二叉树 T 中序号为 e 的子树,PostOrderTraverse()调用
  if(T[2 * e + 1]!= Nil) // 序号为 e 的结点的左子树不空
    PostTraverse(T,2 * e + 1); // 递归后序遍历树 T 中序号为 e 的结点的左子树
```

```
  if(T[2 * e + 2]!= Nil) // 序号为 e 的结点的右子树不空
    PostTraverse(T,2 * e + 2); // 递归后序遍历树 T 中序号为 e 的结点的右子树
  VisitFunc(T[e]); // 访问树 T 中序号为 e 的结点
}
void PostOrderTraverse(SqBiTree T,void(* Visit)(TElemType))
{ // 初始条件：二叉树 T 存在,Visit 是对结点操作的应用函数
  // 操作结果：后序遍历 T,对每个结点调用函数 Visit 一次且仅一次
  VisitFunc = Visit;
  if(!BiTreeEmpty(T)) // 树不空
    PostTraverse(T,0); // 递归后序遍历树 T 中序号为 0 的树(树 T 自身)
  printf("\n");
}
void LevelOrderTraverse(SqBiTree T,void(* Visit)(TElemType))
{ // 层序遍历二叉树 T
  int i = MAX_TREE_SIZE - 1,j;
  while(T[i] == Nil)
    i--; // 找到最后一个非空结点的序号
  for(j = 0;j<= i;j++ ) // 从根结点起,按层序遍历二叉树
    if(T[j]!= Nil)
      Visit(T[j]); // 只遍历非空的结点
  printf("\n");
}
void Print(SqBiTree T)
{ // 逐层、按本层序号输出二叉树 T
  int j,k;
  position p;
  TElemType e;
  for(j = 1;j<= BiTreeDepth(T);j++ ) // j 为当前层
  { printf("第%d 层：",j);
    p.level = j; // 当前结点所在层
    for(k = 1;k<= pow(2,j - 1);k++ )
    { p.order = k; // 当前结点在本层的顺序
      e = Value(T,p); // 该结点的值赋给 e
      if(e!= Nil) // e 非空
        printf("%d:"form" ",k,e); // 输出在本层的顺序及值
    }
    printf("\n");
  }
}
```

```
// func6-1.cpp 利用条件编译,在主程序中选择结点的类型,访问树结点的函数
#include"c1.h"
#if CHAR // CHAR 值为非零,结点类型为字符
  typedef char TElemType; // 定义树元素类型为字符型
  TElemType Nil = ' '; // 设字符型以空格符为空
```

```
  #define form "%c" // 输入输出的格式为%c
#else // CHAR 值为零，结点类型为整型
  typedef int TElemType; // 定义树元素类型为整型
  TElemType Nil = 0; // 设整型以 0 为空
  #define form "%d" // 输入输出的格式为%d
#endif
void visit(TElemType e)
{ printf(form" ",e); // 定义 CHAR 为 1 时，以字符格式输出；CHAR 为 0 时，以整型格式输出
}
```

```
// main6-1.cpp 检验 bo6-1.cpp 的主程序，利用条件编译选择数据类型为 char 或 int
//#define CHAR 1 // 字符型。第 2 行
#define CHAR 0 // 整型(二者选一)。第 3 行
#include"func6-1.cpp" // 利用条件编译，在主程序中选择结点的类型，访问树结点的函数
#include"c6-1.h" // 二叉树的顺序存储结构
#include"bo6-1.cpp" // 二叉树顺序存储结构的基本操作
void main()
{
  Status i;
  int j;
  position p;
  TElemType e;
  SqBiTree T,s;
  InitBiTree(T); // 初始化二叉树 T
  CreateBiTree(T); // 建立二叉树 T
  printf("建立二叉树后，树空否？%d(1：是 0：否)。树的深度=%d。\n",BiTreeEmpty(T),
  BiTreeDepth(T));
  i=Root(T,e); // 将二叉树 T 的根的值赋给 e
  if(i) // 二叉树 T 非空
    printf("二叉树的根为"form"。\n",e);
  else
    printf("树空，无根。\n");
  printf("层序遍历二叉树：\n");
  LevelOrderTraverse(T,visit); // 层序遍历二叉树 T
  printf("中序遍历二叉树：\n");
  InOrderTraverse(T,visit); // 中序遍历二叉树 T
  printf("后序遍历二叉树：\n");
  PostOrderTraverse(T,visit); // 后序遍历二叉树 T
  printf("请输入待修改结点的层号 本层序号：");
  scanf("%d%d",&p.level,&p.order);
  e=Value(T,p); // 将二叉树 T 中位置 p 的结点的值赋给 e
  printf("待修改结点的原值为"form"，请输入新值：",e);
  scanf("%*c"form"%*c",&e);
  Assign(T,p,e); // 将 e 的值赋给二叉树 T 中位置 p 的结点
  printf("先序遍历二叉树：\n");
```

```
    PreOrderTraverse(T,visit); // 先序遍历二叉树 T
    printf("结点"form"的双亲为"form",左右孩子分别为",e,Parent(T,e));
    printf(form"、"form",左右兄弟分别为",LeftChild(T,e),RightChild(T,e));
    printf(form"、"form"。\n",LeftSibling(T,e),RightSibling(T,e));
    InitBiTree(s); // 初始化二叉树 s
    printf("建立右子树为空的树 s:\n");
    CreateBiTree(s); // 建立二叉树 s
    printf("树 s 插到树 T 中,请输入树 T 中树 s 的双亲结点 s 为左(0)或右(1)子树:");
    scanf(form"%d",&e,&j);
    InsertChild(T,e,j,s);
    // 将树 s 插到树 T 中,结点 e 作为树 s 的双亲结点,根据 j 的值确定树 s 是 e 的左或右子树
    Print(T); // 逐层、按本层序号输出二叉树 T
    printf("删除子树,请输入其根结点的双亲的层号 本层序号 其为双亲的左(0)或右(1)子树:");
    scanf("%d%d%d",&p.level,&p.order,&j);
    DeleteChild(T,p,j); // 删除二叉树 T 中位置 p 结点的左(j=0)或右(j=1)子树
    Print(T); // 逐层、按本层序号输出二叉树 T
    ClearBiTree(T); // 清空二叉树 T
    printf("清空二叉树后,树空否?%d(1:是 0:否)。树的深度=%d。\n",BiTreeEmpty(T),
    BiTreeDepth(T));
    i=Root(T,e); // 将二叉树 T 的根的值赋给 e
    if(i) // 二叉树 T 非空
      printf("二叉树的根为"form"。\n",e);
    else // 二叉树 T 为空
      printf("树空,无根。\n");
}
```

**程序运行结果:**

```
请按层序输入结点的值(整型),0 表示空结点,输 999 结束。结点数≤100:
1 2 3 4 5 0 6 7 999↙(见图 6-3)
建立二叉树后,树空否? 0(1:是 0:否)。树的深度=4。
二叉树的根为 1。
层序遍历二叉树:
1 2 3 4 5 6 7
中序遍历二叉树:
7 4 2 5 1 3 6
后序遍历二叉树:
7 4 5 2 6 3 1
请输入待修改结点的层号 本层序号:2 2↙
待修改结点的原值为 3,请输入新值:8↙
先序遍历二叉树:
1 2 4 7 5 8 6
结点 8 的双亲为 1,左右孩子分别为 0、6,左右兄弟分别为 2、0。
建立右子树为空的树 s:
请按层序输入结点的值(整型),0 表示空结点,输 999 结束。结点数≤100:
10 11 0 13 14 0 0 17 999↙(见图 6-4)
```

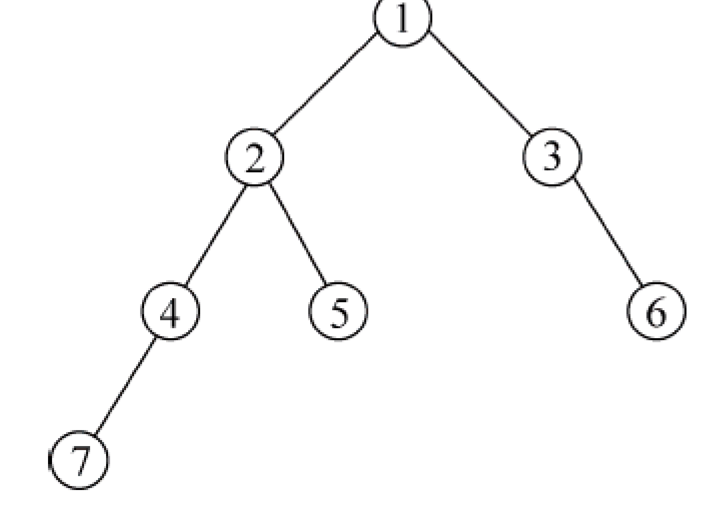



图 6-3 运行 main6-1.cpp 生成的二叉树

```
树 s 插到树 T 中,请输入树 T 中树 s 的双亲结点 s 为左(0)或右(1)子树:2 1↙(见图 6-5)
第 1 层:1:1
第 2 层:1:2 2:8
第 3 层:1:4 2:10 4:6
第 4 层:1:7 3:11 4:5
第 5 层:5:13 6:14
第 6 层:9:17
删除子树,请输入其根结点的双亲的层号 本层序号 其为双亲的左(0)或右(1)子树:3 2 0↙
第 1 层:1:1(见图 6-6)
```

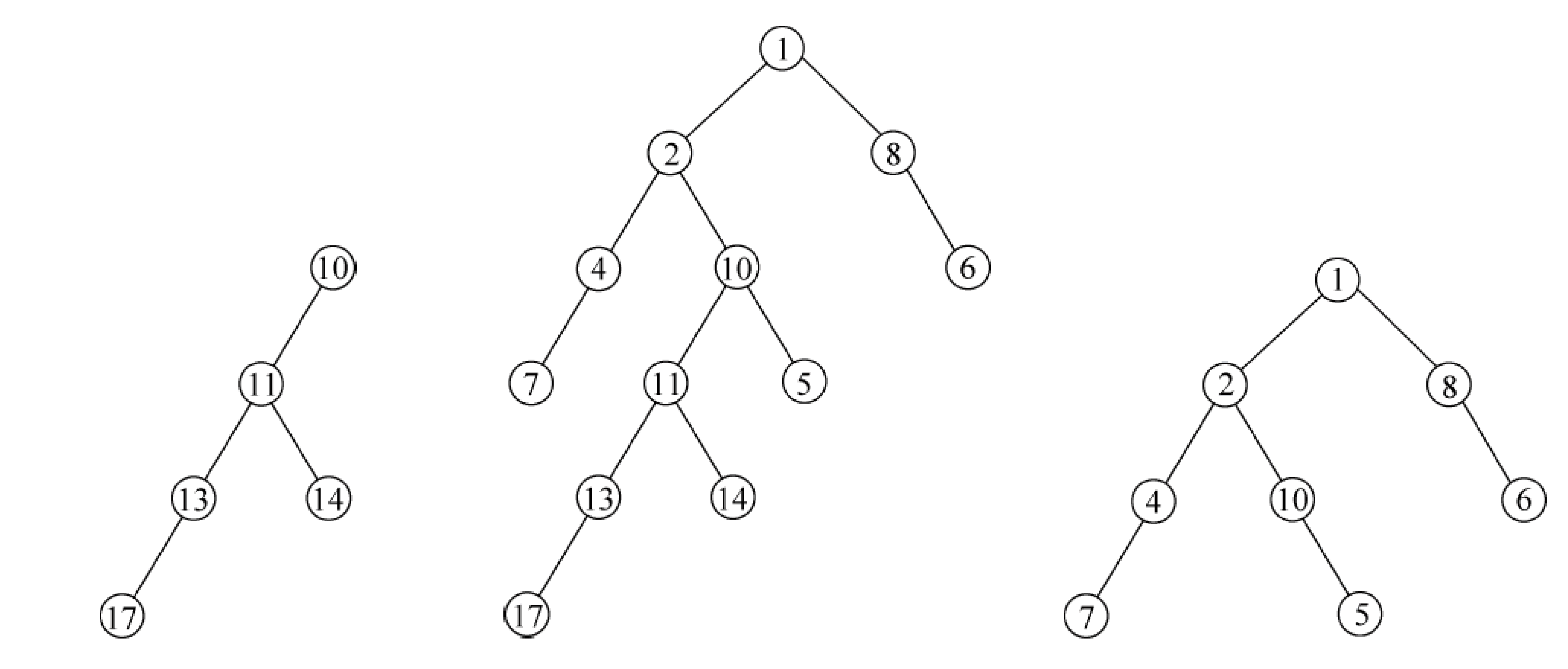


图 6-4　右子树为空的树 s　　图 6-5　插入树 s 后的二叉树　　图 6-6　删除子树后的二叉树

```
第 2 层:1:2 2:8
第 3 层:1:4 2:10 4:6
第 4 层:1:7 4:5
清空二叉树后,树空否? 1(1:是 0:否)。树的深度 = 0。
树空,无根。
```

```
// c6-2.h 二叉树的二叉链表存储结构。在教科书第 127 页(见图 6-7)
typedef struct BiTNode
{ TElemType data; // 结点的值
  BiTNode *lchild, *rchild; // 左右孩子指针
}BiTNode, *BiTree;
```

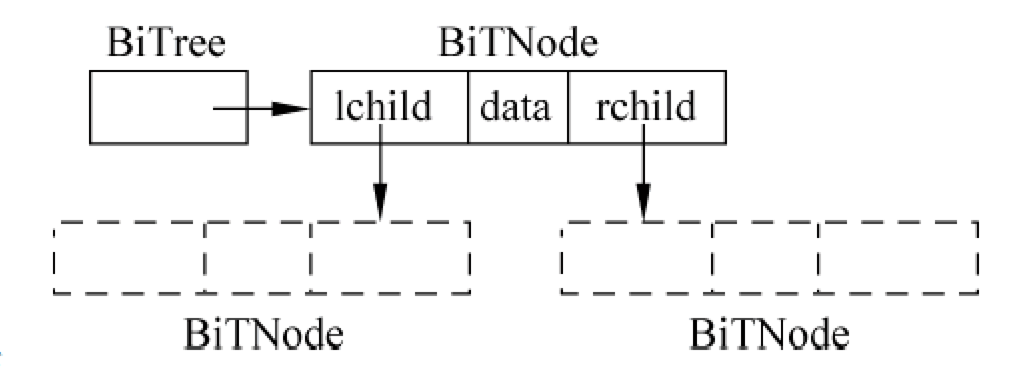



图 6-7　二叉树的二叉链表存储结构

二叉树的二叉链表存储结构删除和插入结点或子树都很灵活。结点动态生成,可充分利用存储空间。图 6-8 是图 6-1(a)所示二叉树的二叉链表存储结构。bo6-2.cpp 和 bo6-3.cpp 是二叉链表存储结构的基本操作,其中,调用按先序次序构造二叉链表的函数 CreateBiTree()(算法 6.4)时,不仅要按先序次序输入结点的值,而且还要把叶子结点的左右孩子指针和度为 1 的结点的空指针输入。其原因是只根据结点的先序次序还不能唯一确定二叉树的形状。如图 6-9 所示,三棵树的先序次序都是 abc。这样,在调用函数 CreateBiTree()时,输入 abc 就会产生多义性。如果把叶子结点的左右孩子指针和度为 1 的结点的空指针(以 ␣ 表示,即空格)也按先序输入,可以唯一确定二叉树的形状。

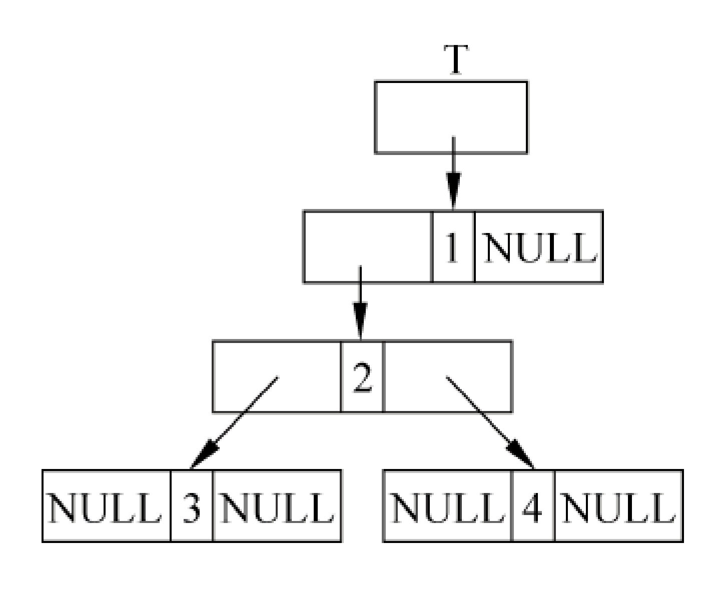



图 6-8 二叉树的二叉链表存储结构(以图 6-1(a)为例)

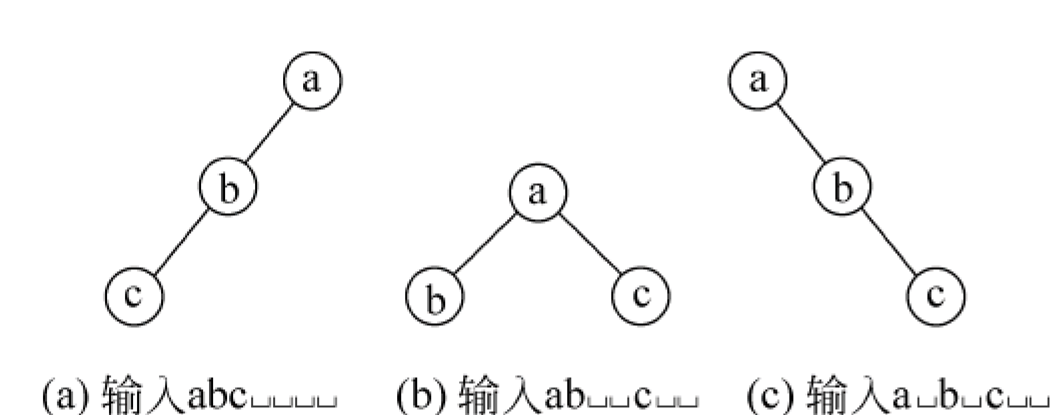



图 6-9 三棵先序遍历都为 abc 的二叉树

二叉链表存储结构的许多基本操作都采用了递归函数,因为二叉树的层数是不定的,正确采用递归函数可简化编程。注意到这些递归函数的特点:一是降阶的,二是有出口的。

```
// bo6-2.cpp 二叉链表的 4 个基本操作,包括算法 6.1,func8-4.cpp 等调用
#define ClearBiTree DestroyBiTree // 清空二叉树和销毁二叉树的操作一样
void InitBiTree(BiTree &T)
{ // 操作结果:构造空二叉树 T(见图 6-10)
  T = NULL;
}
```

T

NULL

图 6-10 空的和销毁的二叉树 T

```
void DestroyBiTree(BiTree &T)
{ // 初始条件:二叉树 T 存在。操作结果:销毁二叉树 T(见图 6-10)
  if(T) // 非空树
  { DestroyBiTree(T->lchild); // 递归销毁左子树,如无左子树,则不执行任何操作
    DestroyBiTree(T->rchild); // 递归销毁右子树,如无右子树,则不执行任何操作
    free(T); // 释放根结点
    T = NULL; // 空指针赋 0
  }
}
void PreOrderTraverse(BiTree T,void(* Visit)(TElemType))
{ // 初始条件:二叉树 T 存在,Visit 是对结点操作的应用函数。修改算法 6.1
  // 操作结果:先序递归遍历 T,对每个结点调用函数 Visit 一次且仅一次
  if(T) // T 不空
  { Visit(T->data); // 先访问根结点
    PreOrderTraverse(T->lchild,Visit); // 再先序遍历左子树
    PreOrderTraverse(T->rchild,Visit); // 最后先序遍历右子树
  }
}
void InOrderTraverse(BiTree T,void(* Visit)(TElemType))
{ // 初始条件:二叉树 T 存在,Visit 是对结点操作的应用函数
  // 操作结果:中序递归遍历 T,对每个结点调用函数 Visit 一次且仅一次
  if(T)
  { InOrderTraverse(T->lchild,Visit); // 先中序遍历左子树
    Visit(T->data); // 再访问根结点
    InOrderTraverse(T->rchild,Visit); // 最后中序遍历右子树
  }
```

```
}

// bo6-3.cpp 二叉链表(存储结构由 c6-2.h 定义)的基本操作(18 个),包括算法 6.2~算法 6.4
Status BiTreeEmpty(BiTree T)
{ // 初始条件：二叉树 T 存在。操作结果：若 T 为空二叉树,则返回 TRUE；否则返回 FALSE
  if(T)
    return FALSE;
  else
    return TRUE;
}
int BiTreeDepth(BiTree T)
{ // 初始条件：二叉树 T 存在。操作结果：返回 T 的深度
  int i,j;
  if(!T)
    return 0; // 空树深度为 0
  i=BiTreeDepth(T->lchild); // i 为左子树的深度,如左子树为空,则 i 为 0
  j=BiTreeDepth(T->rchild); // j 为右子树的深度,如右子树为空,则 j 为 0
  return i>j?i+1:j+1; // T 的深度为其左右子树的深度中的大者+1
}
TElemType Root(BiTree T)
{ // 初始条件：二叉树 T 存在。操作结果：返回 T 的根
  if(BiTreeEmpty(T)) // 二叉树 T 为空
    return Nil; // 返回"空"
  else // 二叉树 T 不空
    return T->data; // 返回根结点的值
}
TElemType Value(BiTree p)
{ // 初始条件：二叉树 T 存在,p 指向 T 中某个结点。操作结果：返回 p 所指结点的值
  return p->data;
}
void Assign(BiTree p,TElemType value)
{ // 给 p 所指结点赋值为 value
  p->data=value;
}
typedef BiTree QElemType; // 定义队列元素为二叉树的指针类型
#include"c3-2.h" // 链队列
#include"bo3-2.cpp" // 链队列的基本操作
BiTree Point(BiTree T,TElemType s)
{ // 返回二叉树 T 中指向元素值为 s 的结点的指针。新增
  LinkQueue q;
  QElemType a;
  if(T) // 非空树
  { InitQueue(q); // 初始化队列
    EnQueue(q,T); // 根指针入队
    while(!QueueEmpty(q)) // 队不空
```

```
    { DeQueue(q,a); // 出队,队列元素赋给 a
      if(a->data == s) // a 所指结点的值为 s
        return a; // 返回 a
      if(a->lchild) // 有左孩子
        EnQueue(q,a->lchild); // 入队左孩子
      if(a->rchild) // 有右孩子
        EnQueue(q,a->rchild); // 入队右孩子
    }
  }
  return NULL; // 二叉树 T 中没有元素值为 s 的结点
}
TElemType LeftChild(BiTree T,TElemType e)
{ // 初始条件:二叉树 T 存在,e 是 T 中某个结点
  // 操作结果:返回 e 的左孩子。若 e 无左孩子,则返回"空"
  BiTree a;
  if(T) // 非空树
  { a = Point(T,e); // a 是指向结点 e 的指针
    if(a&&a->lchild) // T 中存在结点 e 且 e 存在左孩子
      return a->lchild->data; // 返回 e 的左孩子的值
  }
  return Nil; // 其余情况返回"空"
}
TElemType RightChild(BiTree T,TElemType e)
{ // 初始条件:二叉树 T 存在,e 是 T 中某个结点
  // 操作结果:返回 e 的右孩子。若 e 无右孩子,则返回"空"
  BiTree a;
  if(T) // 非空树
  { a = Point(T,e); // a 是指向结点 e 的指针
    if(a&&a->rchild) // T 中存在结点 e 且 e 存在右孩子
      return a->rchild->data; // 返回 e 的右孩子的值
  }
  return Nil; // 其余情况返回空
}
Status DeleteChild(BiTree p,int LR) // 形参 T 无用
{ // 初始条件:二叉树 T 存在,p 指向 T 中某个结点,LR 为 0 或 1
  // 操作结果:根据 LR 为 0 或 1,删除 T 中 p 所指结点的左或右子树
  if(p) // p 不空
  { if(LR == 0) // 删除左子树
      ClearBiTree(p->lchild); // 清空 p 所指结点的左子树
    else // 删除右子树
      ClearBiTree(p->rchild); // 清空 p 所指结点的右子树
    return OK;
  }
  return ERROR; // p 空,返回 ERROR
}
```

```
void PostOrderTraverse(BiTree T,void(* Visit)(TElemType))
{ // 初始条件：二叉树 T 存在,Visit 是对结点操作的应用函数
  // 操作结果：后序递归遍历 T,对每个结点调用函数 Visit 一次且仅一次
  if(T) // T 不空
  { PostOrderTraverse(T->lchild,Visit); // 先后序遍历左子树
    PostOrderTraverse(T->rchild,Visit); // 再后序遍历右子树
    Visit(T->data); // 最后访问根结点
  }
}
void LevelOrderTraverse(BiTree T,void(* Visit)(TElemType))
{ // 初始条件：二叉树 T 存在,Visit 是对结点操作的应用函数
  // 操作结果：层序递归遍历 T(利用队列),对每个结点调用函数 Visit 一次且仅一次
  LinkQueue q;
  QElemType a;
  if(T) // T 不空
  { InitQueue(q); // 初始化队列 q
    EnQueue(q,T); // 根指针入队
    while(!QueueEmpty(q)) // 队列不空
    { DeQueue(q,a); // 出队元素(指针),赋给 a
      Visit(a->data); // 访问 a 所指结点
      if(a->lchild!= NULL) // a 有左孩子
        EnQueue(q,a->lchild); // 入队 a 的左孩子
      if(a->rchild!= NULL) // a 有右孩子
        EnQueue(q,a->rchild); // 入队 a 的右孩子
    }
    printf("\n");
  }
}
void CreateBiTree(BiTree &T)
{ // 算法 6.4：按先序次序输入二叉树中结点的值(可为字符型或整型,在主程中定义),
  // 构造二叉链表表示的二叉树 T。变量 Nil 表示空(子)树。修改
  TElemType ch;
  scanf(form,&ch); // 输入结点的值
  if(ch == Nil) // 结点的值为空
    T = NULL;
  else // 结点的值不为空
  { T = (BiTree)malloc(sizeof(BiTNode)); // 生成根结点
    if(!T)
      exit(OVERFLOW);
    T->data = ch; // 将值赋给 T 所指结点
    CreateBiTree(T->lchild); // 递归构造左子树
    CreateBiTree(T->rchild); // 递归构造右子树
  }
}
TElemType Parent(BiTree T,TElemType e)
```

```
{ // 初始条件：二叉树 T 存在，e 是 T 中某个结点
  // 操作结果：若 e 是 T 的非根结点，则返回它的双亲，否则返回“空”
  LinkQueue q;
  QElemType a;
  if(T) // 非空树
  { InitQueue(q); // 初始化队列
    EnQueue(q,T); // 树根指针入队
    while(!QueueEmpty(q)) // 队不空
    { DeQueue(q,a); // 出队，队列元素赋给 a
      if(a->lchild&&a->lchild->data == e||a->rchild&&a->rchild->data == e)
       // 找到 e(是其左或右孩子)
        return a->data; // 返回 e 的双亲的值
      else // 未找到 e，则入队其左右孩子指针(如果非空)
      { if(a->lchild) // a 有左孩子
          EnQueue(q,a->lchild); // 入队左孩子指针
        if(a->rchild) // a 有右孩子
          EnQueue(q,a->rchild); // 入队右孩子指针
      }
    }
  }
  return Nil; // 树空或未找到 e
}
TElemType LeftSibling(BiTree T,TElemType e)
{ // 初始条件：二叉树 T 存在，e 是 T 中某个结点
  // 操作结果：返回 e 的左兄弟。若 e 是 T 的左孩子或无左兄弟，则返回“空”
  TElemType a;
  BiTree p;
  if(T) // 非空树
  { a = Parent(T,e); // a 为 e 的双亲
    if(a!= Nil) // 找到 e 的双亲
    { p = Point(T,a); // p 为指向结点 a 的指针
      if(p->lchild&&p->rchild&&p->rchild->data == e) // p 存在左右孩子且右孩子是 e
        return p->lchild->data; // 返回 p 的左孩子(e 的左兄弟)
    }
  }
  return Nil; // 其余情况返回空
}
TElemType RightSibling(BiTree T,TElemType e)
{ // 初始条件：二叉树 T 存在，e 是 T 中某个结点
  // 操作结果：返回 e 的右兄弟。若 e 是 T 的右孩子或无右兄弟，则返回“空”
  TElemType a;
  BiTree p;
  if(T) // 非空树
  { a = Parent(T,e); // a 为 e 的双亲
```

```
    if(a!=Nil) // 找到 e 的双亲
    { p=Point(T,a); // p 为指向结点 a 的指针
      if(p->lchild&&p->rchild&&p->lchild->data==e) // p 存在左右孩子且左孩子是 e
        return p->rchild->data; // 返回 p 的右孩子(e 的右兄弟)
    }
  }
  return Nil; // 其余情况返回空
}
Status InsertChild(BiTree p,int LR,BiTree c) // 形参 T 无用
{ // 初始条件:二叉树 T 存在,p 指向 T 中某个结点,LR 为 0 或 1,
  //           非空二叉树 c 与 T 不相交且右子树为空
  // 操作结果:根据 LR 为 0 或 1,插入 c 为 T 中 p 所指结点的左或右子树。p 所指结点的
  //           原有左或右子树则成为 c 的右子树
  if(p) // p 不空
  { if(LR==0) // 把二叉树 c 插为 p 所指结点的左子树
    { c->rchild=p->lchild; // p 所指结点的原有左子树成为 c 的右子树
      p->lchild=c; // 二叉树 c 成为 p 的左子树
    }
    else // LR==1,把二叉树 c 插为 p 所指结点的右子树
    { c->rchild=p->rchild; // p 所指结点的原有右子树成为 c 的右子树
      p->rchild=c; // 二叉树 c 成为 p 的右子树
    }
    return OK;
  }
  return ERROR; // p 空
}
typedef BiTree SElemType; // 定义栈元素为二叉树的指针类型
#include"c3-1.h" // 顺序栈
#include"bo3-1.cpp" // 顺序栈的基本操作
void InOrderTraverse1(BiTree T,void(*Visit)(TElemType))
{ // 采用二叉链表存储结构,Visit 是对数据元素操作的应用函数。修改算法 6.3
  // 中序遍历二叉树 T 的非递归算法(利用栈),对每个数据元素调用函数 Visit
  SqStack S;
  InitStack(S); // 初始化栈 S
  while(T||!StackEmpty(S)) // 当二叉树 T 不空或者栈不空
  { if(T) // 二叉树 T 不空
    { // 根指针进栈,遍历左子树
      Push(S,T); // 入栈根指针
      T=T->lchild; // T 指向其左孩子
    }
    else // 根指针退栈,访问根结点,遍历右子树
    { Pop(S,T); // 出栈根指针
      Visit(T->data); // 访问根结点
      T=T->rchild; // T 指向其右孩子
```

```
    }
  }
  printf("\n");
}
void InOrderTraverse2(BiTree T,void(* Visit)(TElemType))
{ // 采用二叉链表存储结构,Visit 是对数据元素操作的应用函数。算法 6.2,有改动
  // 中序遍历二叉树 T 的非递归算法(利用栈),对每个数据元素调用函数 Visit
  SqStack S;
  BiTree p;
  InitStack(S); // 初始化栈 S
  Push(S,T); // 根指针进栈(无论空否)
  while(!StackEmpty(S)) // 栈不空
  { while(GetTop(S,p)&&p) // 栈顶元素不为空指针
      Push(S,p->lchild); // 向左走到尽头,入栈左孩子指针
    Pop(S,p); // 空指针退栈,退掉最后入栈的空指针
    if(!StackEmpty(S)) // 访问结点,向右一步
    { Pop(S,p); // 弹出栈顶元素(非空指针)
      Visit(p->data); // 访问刚弹出的结点(目前栈顶元素的左孩子)
      Push(S,p->rchild); // 入栈其右孩子指针
    }
  }
  printf("\n");
}
```

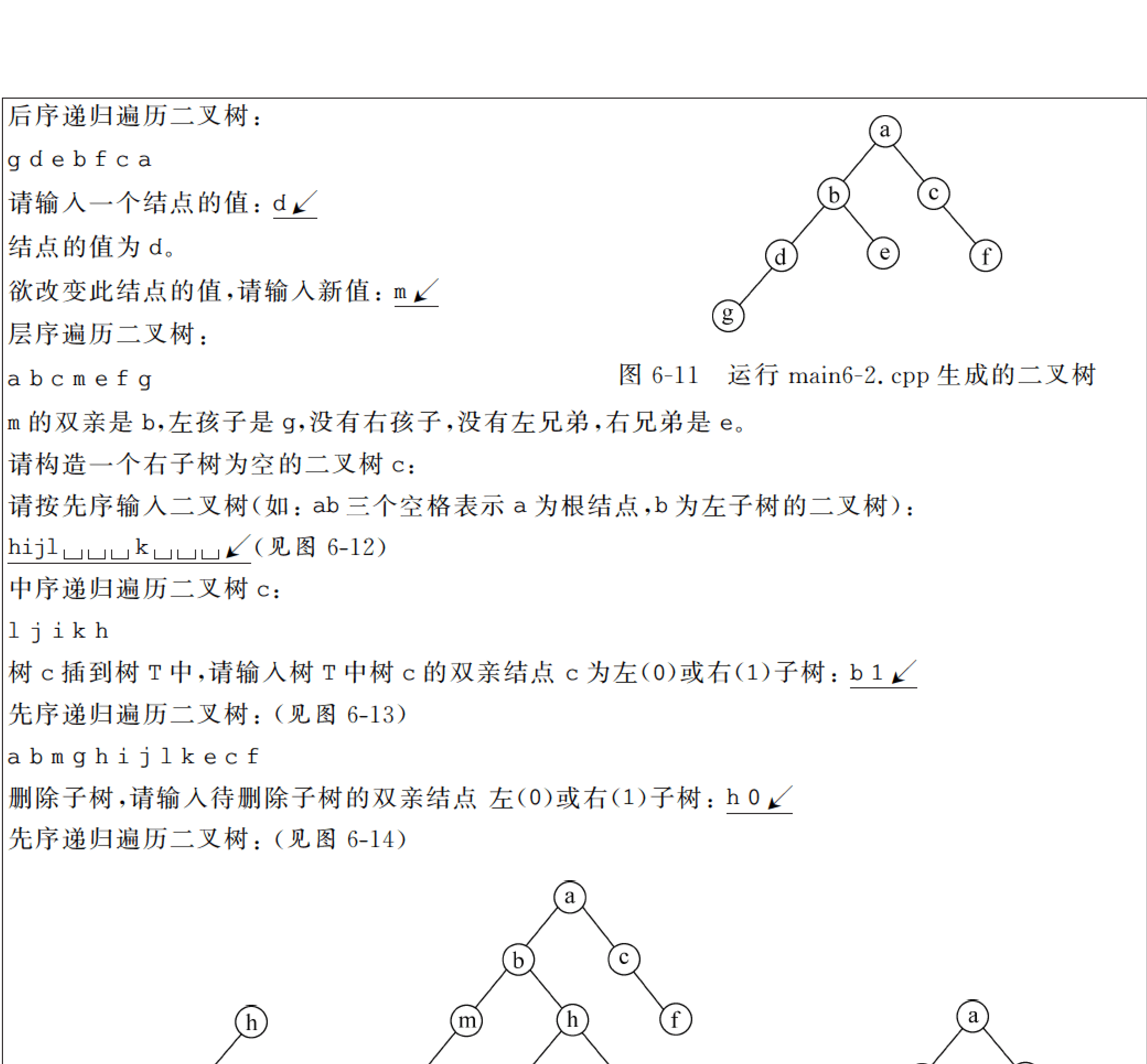
```
// main6-2.cpp 检验二叉链表基本操作的主程序
#define CHAR 1 // 字符型。第 2 行
//#define CHAR 0 // 整型(二者选一)。第 3 行
#include"func6-1.cpp" // 利用条件编译,在主程序中选择结点的类型,访问树结点的函数
#include"c6-2.h" // 二叉树的二叉链表存储结构
#include"bo6-2.cpp" // 二叉树的二叉链表存储结构的 4 个基本操作,func8-4.cpp 等调用
#include"bo6-3.cpp" // 二叉树的二叉链表存储结构的 18 个基本操作
void main()
{
  int i;
  BiTree T,p,c;
  TElemType e1,e2;
  InitBiTree(T); // 初始化二叉树 T
  printf("构造空二叉树后,树空否?%d(1:是 0:否)。树的深度=%d。\n",BiTreeEmpty(T),
  BiTreeDepth(T));
  e1=Root(T); // e1 为二叉树 T 的根结点的值
  if(e1!=Nil)
    printf("二叉树的根为"form"。\n",e1);
  else
    printf("树空,无根。\n");
```

```
#if CHAR // CHAR 值为非零,结点类型为字符
  printf("请按先序输入二叉树(如：ab 三个空格表示 a 为根结点,b 为左子树的二叉树)：\n");
#else // CHAR 值为零,结点类型为整型
  printf("请按先序输入二叉树(如：1 2 0 0 0 表示 1 为根结点,2 为左子树的二叉树)：\n");
#endif
  CreateBiTree(T); // 建立二叉树 T
  printf("建立二叉树后,树空否？%d(1：是 0：否)。树的深度 = %d。\n",BiTreeEmpty(T),
  BiTreeDepth(T));
  e1 = Root(T); // e1 为二叉树 T 的根结点的值
  if(e1 != Nil)
    printf("二叉树的根为"form"。\n",e1);
  else
    printf("树空,无根。\n");
  printf("中序递归遍历二叉树：\n");
  InOrderTraverse(T,visit); // 中序递归遍历二叉树 T
  printf("\n 后序递归遍历二叉树：\n");
  PostOrderTraverse(T,visit); // 后序递归遍历二叉树 T
  printf("\n 请输入一个结点的值：");
  scanf("% * c"form,&e1);
  p = Point(T,e1); // p 指向为 e1 的指针
  printf("结点的值为"form"。\n",Value(p));
  printf("欲改变此结点的值,请输入新值：");
  scanf("% * c"form"% * c",&e2); // 后一个 % * c 吃掉回车符,为调用 CreateBiTree()做准备
  Assign(p,e2); // 将 e2 的值赋给 p 所指结点,代替 e1
  printf("层序遍历二叉树：\n");
  LevelOrderTraverse(T,visit); // 层序递归遍历二叉树 T
  e1 = Parent(T,e2); // 将二叉树 T 中结点 e2 的双亲的值赋给 e1
  if(e1 != Nil)
    printf(form"的双亲是"form",",e2,e1);
  else
    printf(form"没有双亲,",e2);
  e1 = LeftChild(T,e2); // 将二叉树 T 中结点 e2 的左孩子的值赋给 e1
  if(e1 != Nil)
    printf("左孩子是"form",",e1);
  else
    printf("没有左孩子,");
  e1 = RightChild(T,e2); // 将二叉树 T 中结点 e2 的右孩子的值赋给 e1
  if(e1 != Nil)
    printf("右孩子是"form",",e1);
  else
    printf("没有右孩子,");
  e1 = LeftSibling(T,e2); // 将二叉树 T 中结点 e2 的左兄弟的值赋给 e1
  if(e1 != Nil)
    printf("左兄弟是"form",",e1);
```

```
    else
      printf("没有左兄弟,");
    e1 = RightSibling(T,e2); // 将二叉树 T 中结点 e2 的右兄弟的值赋给 e1
    if(e1 != Nil)
      printf("右兄弟是"form"。\n",e1);
    else
      printf("没有右兄弟。\n");
    InitBiTree(c); // 初始化二叉树 c
    printf("请构造一个右子树为空的二叉树 c:\n");
#if CHAR // CHAR 值为非零,结点类型为字符
    printf("请按先序输入二叉树(如:ab 三个空格表示 a 为根结点,b 为左子树的二叉树):\n");
#else // CHAR 值为零,结点类型为整型
    printf("请按先序输入二叉树(如:1 2 0 0 0 表示 1 为根结点,2 为左子树的二叉树):\n");
#endif
    CreateBiTree(c); // 建立二叉树 c
    printf("中序递归遍历二叉树 c:\n");
    InOrderTraverse(c,visit); // 中序递归遍历二叉树 c
    printf("\n 树 c 插到树 T 中,请输入树 T 中树 c 的双亲结点 c 为左(0)或右(1)子树:");
    scanf("% * c"form"%d",&e1,&i);
    p = Point(T,e1); // p 指向二叉树 T 中将作为二叉树 c 的双亲结点的 e1
    InsertChild(p,i,c); // 将树 c 插入到二叉树 T 中作为结点 e1(p 所指)的左(i = 0)或右(i = 1)子树
    printf("先序递归遍历二叉树:\n");
    PreOrderTraverse(T,visit); // 先序递归遍历二叉树 T
    printf("\n 删除子树,请输入待删除子树的双亲结点 左(0)或右(1)子树:");
    scanf("% * c"form"%d",&e1,&i);
    p = Point(T,e1); // p 指向二叉树 T 中待删除子树的双亲结点 e1
    DeleteChild(p,i); // 删除 p 所指结点(e1)的左(i = 0)或右(i = 1)子树
    printf("先序递归遍历二叉树:\n");
    PreOrderTraverse(T,visit); // 先序递归遍历二叉树 T
    printf("\n 中序非递归遍历二叉树:\n");
    InOrderTraverse1(T,visit); // 中序非递归遍历二叉树 T
    printf("中序非递归遍历二叉树(另一种方法):\n");
    InOrderTraverse2(T,visit); // 中序非递归遍历二叉树 T(另一种方法)
    DestroyBiTree(T); // 销毁二叉树 T
}
```

**程序运行结果:**(输入中的␣表示空格)

```
构造空二叉树后,树空否? 1(1:是 0:否)。树的深度 = 0。
树空,无根。
请按先序输入二叉树(如:ab 三个空格表示 a 为根结点,b 为左子树的二叉树):
abdg␣␣␣e␣␣c␣f␣␣↙(见图 6-11)
建立二叉树后,树空否? 0(1:是 0:否)。树的深度 = 4。
二叉树的根为 a。
中序递归遍历二叉树:
g d b e a c f
```




后序递归遍历二叉树：
g d e b f c a
请输入一个结点的值：d↙
结点的值为 d。
欲改变此结点的值，请输入新值：m↙
层序遍历二叉树：
a b c m e f g

图 6-11　运行 main6-2.cpp 生成的二叉树

m 的双亲是 b，左孩子是 g，没有右孩子，没有左兄弟，右兄弟是 e。
请构造一个右子树为空的二叉树 c：
请按先序输入二叉树(如：ab 三个空格表示 a 为根结点，b 为左子树的二叉树)：
hijl␣␣␣k␣␣␣↙(见图 6-12)
中序递归遍历二叉树 c：
l j i k h
树 c 插到树 T 中，请输入树 T 中树 c 的双亲结点 c 为左(0)或右(1)子树：b 1↙
先序递归遍历二叉树：(见图 6-13)
a b m g h i j l k e c f
删除子树，请输入待删除子树的双亲结点 左(0)或右(1)子树：h 0↙
先序递归遍历二叉树：(见图 6-14)

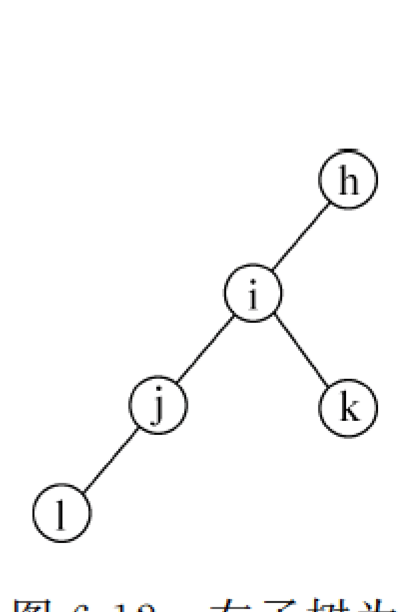



图 6-12　右子树为空的树 c

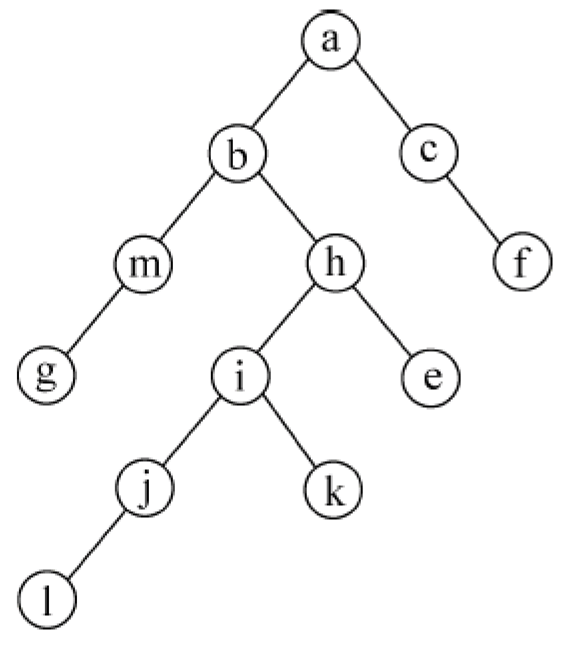



图 6-13　树 c 插到树 T 中作为结点 b 的右子树

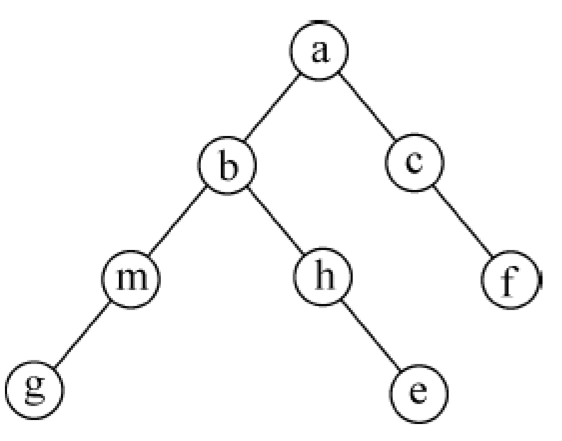



图 6-14　删除 h 的左子树后的二叉树

a b m g h e c f
中序非递归遍历二叉树：
g m b h e a c f
中序非递归遍历二叉树(另一种方法)：
g m b h e a c f

## 6.2　遍历二叉树和线索二叉树

### 6.2.1　遍历二叉树

遍历二叉树就是按某种规则，对二叉树的每个结点均访问一次，而且仅访问一次。这实际上就是将非线性的二叉树结构线性化。遍历二叉树的方法有先序、中序、后序和层序 4

种,访问的顺序各不相同。以图 6-1(a)所示二叉树为例,先序遍历的顺序为 1 2 3 4;中序遍历的顺序为 3 2 4 1;后序遍历的顺序为 3 4 2 1;层序遍历的顺序为 1 2 3 4。对于这棵二叉树,层序遍历和先序遍历的顺序碰巧一致。有关二叉链表存储结构生成二叉树和中序非递归遍历二叉树的算法 6.2~算法 6.4 在 bo6-3.cpp 中,算法 6.1 在 bo6-2.cpp 中。

### 6.2.2 线索二叉树

为了更方便、更快捷地遍历二叉树,最好在二叉树的结点上增加 2 个指针,它们分别指向遍历二叉树时该结点的前驱和后继结点。这样,从二叉树的任一结点都可以方便地找到其他结点。但这样做大大降低了结构的存储密度。另外,根据二叉树的性质 3(教科书 124 页),有:$n_0=n_2+1$。空链域$=2n_0+n_1$(叶子结点有 2 个空链域,度为 1 的结点有 1 个空链域)$=n_0+n_1+n_2+1=n+1$。也就是说,在由 n 个结点组成的二叉树中,有 n+1 个指针是空指针。如果能利用这 n+1 个空指针,使它们指向结点的前驱(当左孩子指针空)或后继(当右孩子指针空),则既可不降低结构的存储密度,又可更方便、更快捷地遍历二叉树。不过,这样就无法区别左右孩子指针所指的到底是结点的左右孩子,还是结点的前驱后继了。为了有所区别,另增加 2 个域 LTag 和 RTag。当所指的是孩子,其值为 0(Link);当所指的是前驱后继,其值为 1(Thread)。这样做,结构的存储密度也有所降低,但不大。因为 LTag 和 RTag 分别只占 1 个比特(二进制位)即可。c6-3.h 是二叉树的二叉线索存储结构。它只是比二叉链表存储结构(在 c6-2.h 中)多了 LTag 和 RTag 两个域。

```
// c6-3.h 二叉树的二叉线索存储结构。在教科书第 133 页(见图 6-15)
enum PointerTag // 枚举
{Link,Thread}; // Link(0):指针,Thread(1):线索
struct BiThrNode
{ TElemType data; // 结点的值
  BiThrNode *lchild,*rchild; // 左右孩子指针
  PointerTag LTag:2; // 左标志,占 2bit,修改
  PointerTag RTag:2; // 右标志,占 2bit,修改
};
typedef BiThrNode *BiThrTree;
```

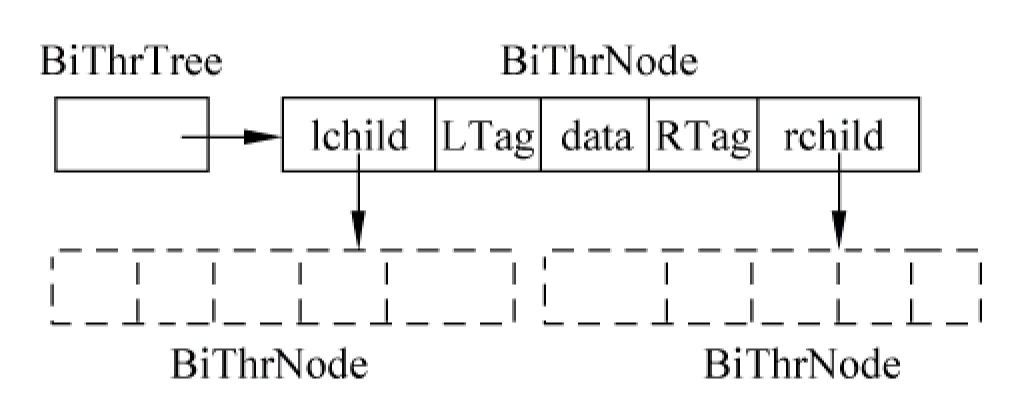


图 6-15 二叉树的二叉线索存储结构

构造线索二叉树的方法和构造以二叉链表存储的二叉树方法相似,都是按先序输入结点的值来构造二叉树的。对比 bo6-3.cpp 中的 CreateBiTree()函数和 bo6-4.cpp 中的 CreateBiThrTree()函数可见,它们的区别有两点:

(1) 二叉树结点的结构不同;

(2) 构造线索二叉树时,若有左右孩子结点,还要给左右标志赋值 0(Link)。

图 6-1(a)所示二叉树调用 CreateBiThrTree()函数产生的二叉树结构如图 6-16 所示。和调用 CreateBiTree()函数产生的二叉树结构(见图 6-8)相比,前者只是多了 LTag 和 RTag 两个域。并且当其相应孩子指针不空时,赋值 0。

调用 CreateBiThrTree()函数,只是构造了一棵可以线索化的二叉树,还没有完成线索化。

因为对于一棵给定的二叉树,其先序、中序、后序和层序遍历的顺序是不同的。显然,其

线索化的操作和遍历的操作也是不同的。

图 6-17 是图 6-1(a)所示二叉树的中序线索二叉树存储结构示例，和二叉链表(见图 6-8)存储结构相比，第一，它多了一个头结点。其左孩子指针指向根结点，右孩子指针(线索)指向中序遍历所访问的最后一个结点。第二，它每个结点的左右孩子指针都不是空指针。在没有孩子的情况下，分别指向该结点中序遍历的前驱或后继。第三，中序遍历的第 1 个结点(最左边的结点，它没有左孩子，本例是结点 3)的左孩子指针(线索)和最后 1 个结点(最右边的结点，它没有右孩子，本例是结点 1)的右孩子指针(线索)都指向头结点。其目的是标志遍历的起点和终点。图 6-18 是空线索二叉树。

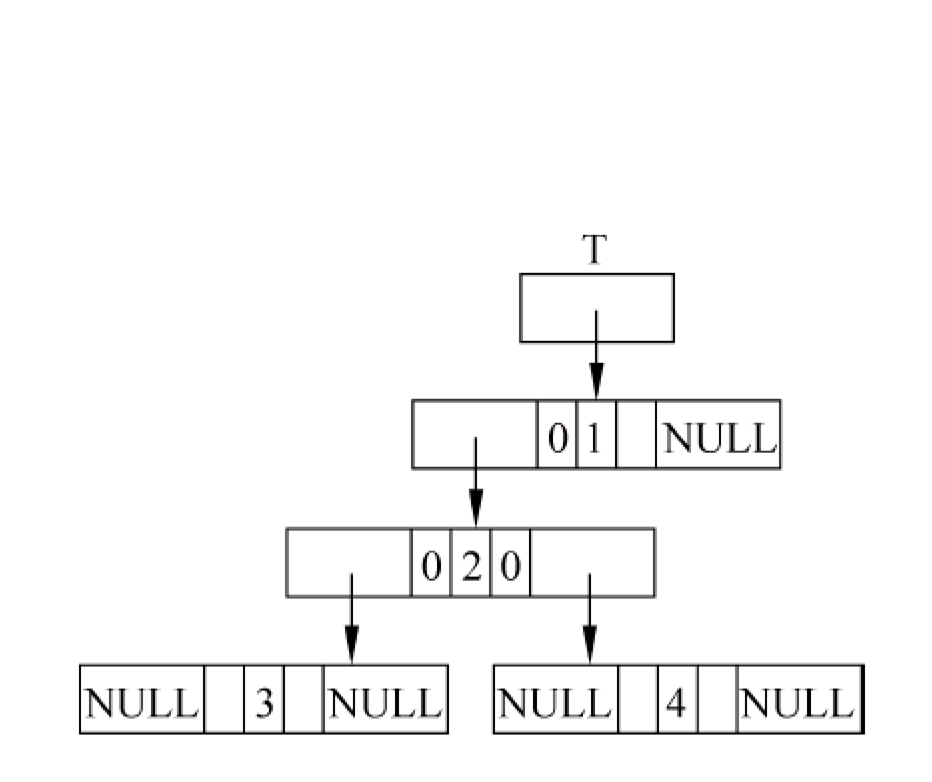



图 6-16　CreateBiThrTree()产生的二叉树(以图 6-1(a)为例)

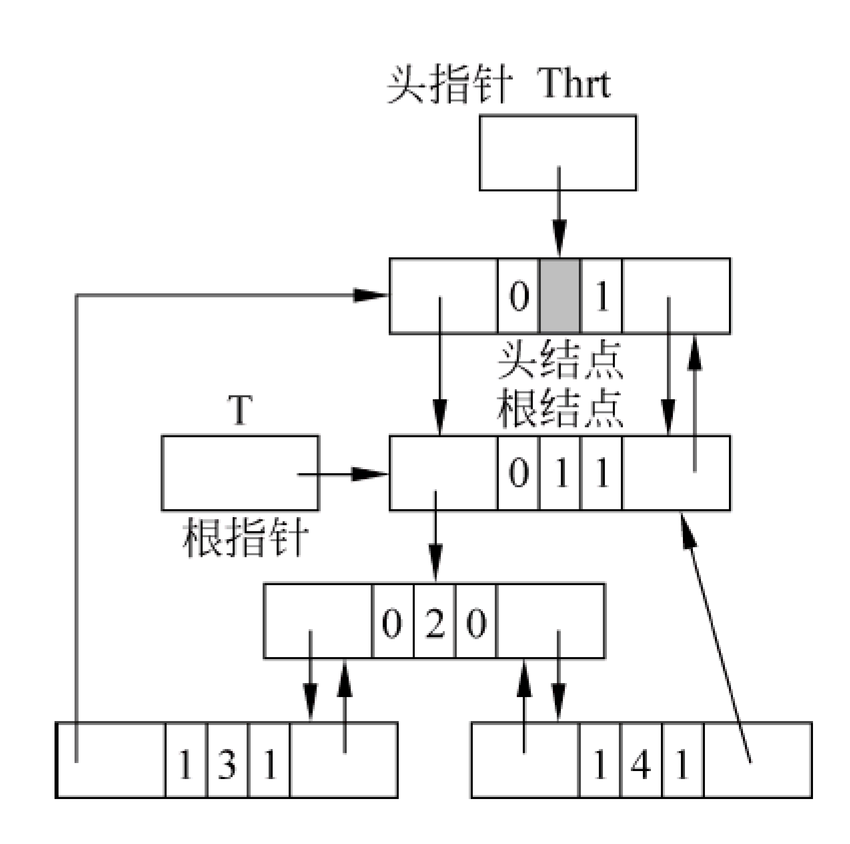



图 6-17　中序线索二叉树存储结构(以图 6-1(a)为例)

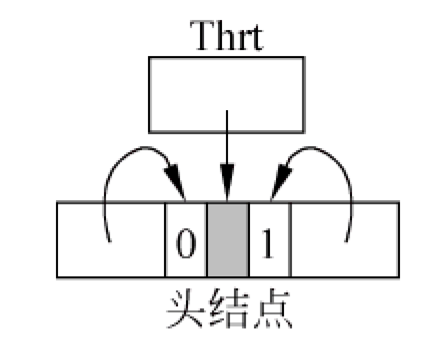



图 6-18　空线索二叉树

bo6-4.cpp 中的 InThreading()函数和 InOrderThreading()函数共同完成了对二叉树的中序遍历线索化。其算法是：设置全局指针变量 pre(之所以设为全局变量，是因为在递归函数 InThreading()和 InOrderThreading()中都要用到，设为全局变量就不必频繁传递变量的值)，令 pre 总是指向遍历的前驱结点，p 指向当前结点；在中序遍历过程中，如果 p 所指结点没有左孩子，则结点的左孩子指针指向 pre 所指结点，结点的 LTag 域的值为 1(Thread)；如果 pre 所指结点没有右孩子，则结点的右孩子指针指向 p，结点的 RTag 域的值为 1(Thread)。

对于图 6-17 所示的中序线索二叉树，我们能不能在找到遍历的第 1 个结点后，顺着右孩子指针一直找到遍历的最后 1 个结点呢？这是不一定的。因为结点的右孩子指针并不一定指向后继结点，它也可能指向右孩子，而右孩子并不一定恰好是后继结点。

bo6-4.cpp 中的 InOrderTraverse_Thr()函数完成了对中序线索二叉树的中序遍历操作。其算法是：当树不空时，由树根向左找，一直找到没有左孩子的结点(最左结点)。这就是中序遍历的第 1 个结点。若该结点没有右孩子，则右孩子指针指向其后继结点；否则，以其右孩子为子树的根，向左找，一直找到没有左孩子的结点。这就是后继结点。当结点的右孩子指针指向头结点，遍历结束。

图 6-19 是图 6-1(a)所示二叉树的先序线索二叉树存储结构示例。其头结点的左孩子指针仍指向根结点，右孩子指针指向先序遍历所访问的最后一个结点。bo6-4.cpp 中的先序线索化的递归函数 PreThreading()与中序线索化的递归函数 InThreading()很相像，都是

利用递归进行线索化，只不过顺序不同。但由于 PreThreading()是先序线索化，所以判断结点是否有左右孩子就不能由其左右孩子指针是否为空决定，而要由结点的 LTag 和 RTag 域是否为 0(Link)来决定。

对于先序线索化二叉树的先序遍历算法是这样的：根结点是遍历的第 1 个结点；如果结点有左孩子，则左孩子是其后继；若结点没有左孩子，则右孩子指针所指的结点是其后继(无论该结点有没有右孩子)。相关程序见 bo6-4.cpp 中的 PreOrderTraverse_Thr()函数。

bo6-4.cpp 中的后序线索化的递归函数 PostThreading()与中序线索化的递归函数 InThreading()也很相像，也是利用递归进行线索化，也只是顺序不同。图 6-20 是图 6-1(a)所示二叉树的后序线索二叉树存储结构示例。其头结点的左右孩子指针都指向根结点。后序遍历的第 1 个结点必定是叶子结点，它的左孩子指针指向头结点。对于后序线索化二叉树的后序遍历算法较复杂。因为根结点是在最后遍历，所以要采用带有双亲指针的三叉链表结构才行。

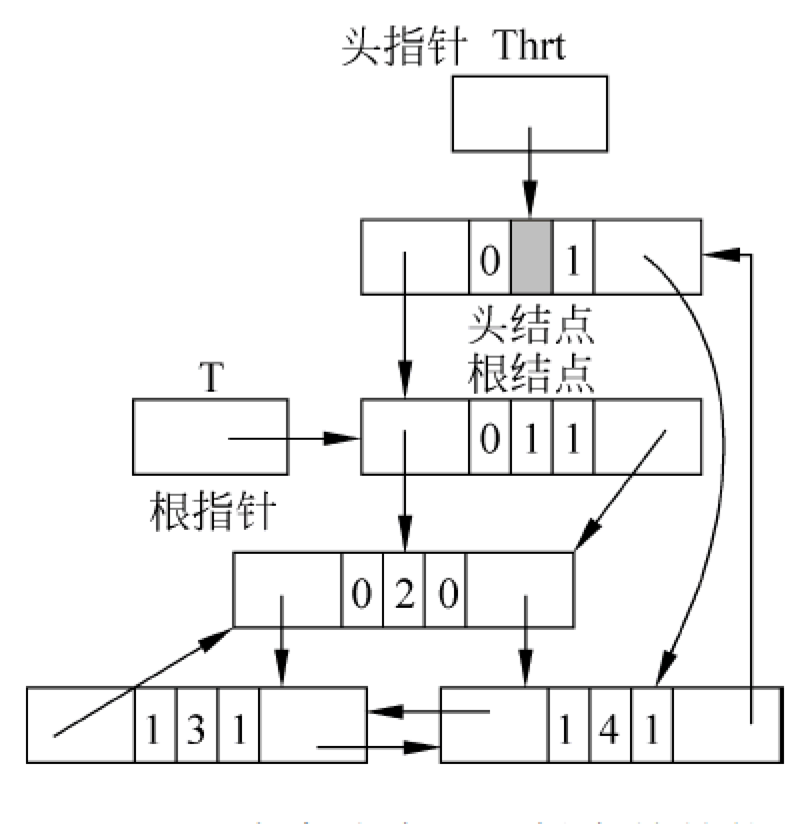



图 6-19 先序线索二叉树存储结构(以图 6-1(a)为例)

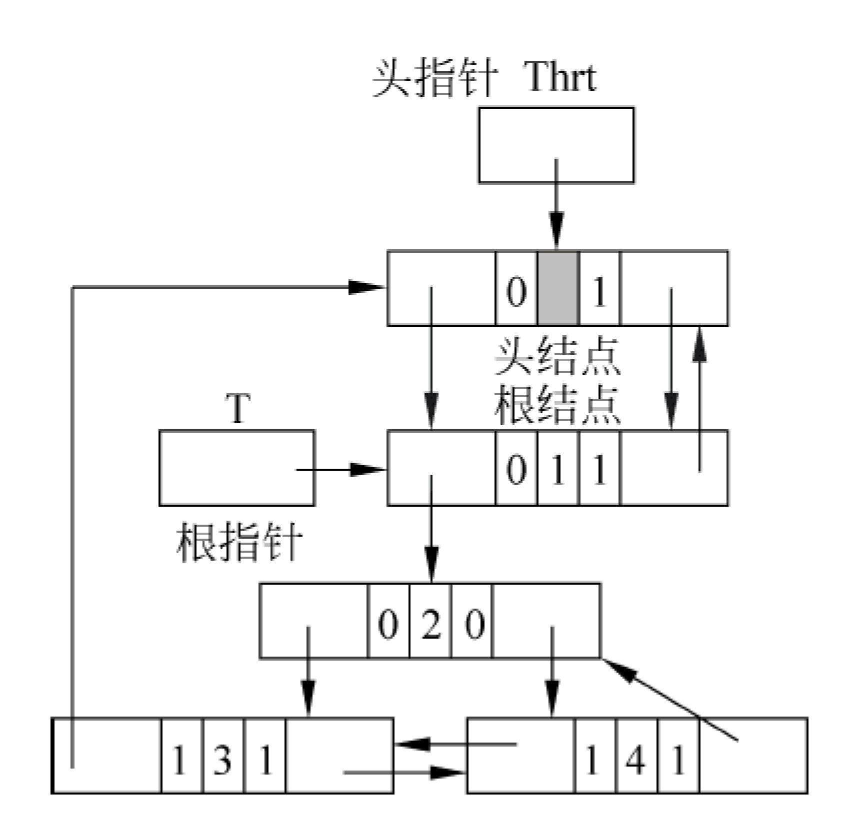



图 6-20 后序线索二叉树存储结构(以图 6-1(a)为例)

```
// bo6-4.cpp 构造线索二叉树的 11 个基本操作，包括算法 6.5～算法 6.7
void CreateBiThrTree(BiThrTree &T)
{ // 按先序输入线索二叉树中结点的值，构造线索二叉树 T。0(整型)/空格(字符型)表示空结点
  TElemType ch;
  scanf(form,&ch); // 输入结点的值
  if(ch == Nil) // 结点的值为空
    T = NULL;
  else // 结点的值不为空
  { T = (BiThrTree)malloc(sizeof(BiThrNode)); // 生成根结点(先序)
    if(!T)
      exit(OVERFLOW);
    T->data = ch; // 将值赋给 T 所指结点
    CreateBiThrTree(T->lchild); // 递归构造左子树
    if(T->lchild) // 有左孩子
      T->LTag = Link; // 给左标志赋值(指针)
    CreateBiThrTree(T->rchild); // 递归构造右子树
```

```
    if(T->rchild) // 有右孩子
      T->RTag = Link; // 给右标志赋值(指针)
  }
}
BiThrTree pre; // 全局变量,始终指向刚刚访问过的结点
void InThreading(BiThrTree p)
{ // 通过中序遍历进行中序线索化,线索化之后 pre 指向最后一个结点。算法 6.7
  if(p) // 线索二叉树不空
  { InThreading(p->lchild); // 递归左子树线索化
    if(!p->lchild) // 没有左孩子
    { p->LTag = Thread; // 左标志为线索(前驱)
      p->lchild = pre; // 左孩子指针指向前驱
    }
    if(!pre->rchild) // 前驱没有右孩子
    { pre->RTag = Thread; // 前驱的右标志为线索(后继)
      pre->rchild = p; // 前驱右孩子指针指向其后继(当前结点 p)
    }
    pre = p; // 保持 pre 指向 p 的前驱
    InThreading(p->rchild); // 递归右子树线索化
  }
}
void InOrderThreading(BiThrTree &Thrt,BiThrTree T)
{ // 中序遍历二叉树 T,并将其中序线索化,Thrt 指向头结点。算法 6.6
  if(!(Thrt = (BiThrTree)malloc(sizeof(BiThrNode)))) // 生成头结点不成功
    exit(OVERFLOW);
  Thrt->LTag = Link; // 建头结点,左标志为指针
  Thrt->RTag = Thread; // 右标志为线索
  Thrt->rchild = Thrt; // 右孩子指针回指
  if(!T) // 若二叉树 T 空,则左孩子指针回指
    Thrt->lchild = Thrt;
  else // 二叉树 T 非空
  { Thrt->lchild = T; // 头结点的左孩子指针指向根结点
    pre = Thrt; // pre(前驱)的初值指向头结点
    InThreading(T); // 中序遍历进行中序线索化,pre 指向中序遍历的最后一个结点
    pre->rchild = Thrt; // 最后一个结点的右孩子指针指向头结点
    pre->RTag = Thread; // 最后一个结点的右标志为线索
    Thrt->rchild = pre; // 头结点的右孩子指针指向中序遍历的最后一个结点
  }
}
void InOrderTraverse_Thr(BiThrTree T,void(*Visit)(TElemType))
{ // 中序遍历线索二叉树 T(头结点)的非递归算法。算法 6.5
  BiThrTree p;
  p = T->lchild; // p 指向根结点
  while(p != T)
  { // 空树或遍历结束时,p == T
    while(p->LTag == Link) // 由根结点一直找到二叉树的最左结点
```

```
      p = p->lchild; // p指向其左孩子
    Visit(p->data); // 访问此结点
    while(p->RTag == Thread&&p->rchild!= T)
    { // p->rchild是线索(后继),且不是遍历的最后一个结点
      p = p->rchild; // p指向其后继
      Visit(p->data); // 访问后继结点
    }
    // 若p->rchild不是线索(是右孩子),p指向右孩子,返回循环,找这棵子树中序遍历的第1个结点
    p = p->rchild;
  }
}
void PreThreading(BiThrTree p)
{ // PreOrderThreading()调用的递归函数
  if(!pre->rchild) // p的前驱没有右孩子
  { pre->RTag = Thread; // pre的右孩子指针为线索
    pre->rchild = p; // p前驱的后继指向p
  }
  if(!p->lchild) // p没有左孩子
  { p->LTag = Thread; // p的左孩子指针为线索
    p->lchild = pre; // p的左孩子指针指向前驱
  }
  pre = p; // 移动前驱
  if(p->LTag == Link) // p有左孩子
    PreThreading(p->lchild); // 对p的左孩子递归调用preThreading()
  if(p->RTag == Link) // p有右孩子
    PreThreading(p->rchild); // 对p的右孩子递归调用preThreading()
}
void PreOrderThreading(BiThrTree &Thrt,BiThrTree T)
{ // 先序线索化二叉树T,头结点的右孩子指针指向先序遍历的最后1个结点
  if(!(Thrt = (BiThrTree)malloc(sizeof(BiThrNode)))) // 生成头结点
    exit(OVERFLOW);
  Thrt->LTag = Link; // 头结点的左孩子指针为孩子
  Thrt->RTag = Thread; // 头结点的右孩子指针为线索
  Thrt->rchild = Thrt; // 头结点的右孩子指针指向自身
  if(!T) // 空树
    Thrt->lchild = Thrt; // 头结点的左孩子指针也指向自身
  else // 非空树
  { Thrt->lchild = T; // 头结点的左孩子指针指向根结点
    pre = Thrt; // 前驱为头结点
    PreThreading(T); // 从头结点开始先序递归线索化
    pre->RTag = Thread; // 最后一个结点的右孩子指针为线索
    pre->rchild = Thrt; // 最后一个结点的后继指向头结点
    Thrt->rchild = pre; // 头结点的后继指向最后一个结点
  }
}
```

```
void PreOrderTraverse_Thr(BiThrTree T,void(* Visit)(TElemType))
{ // 先序遍历线索二叉树 T(头结点)的非递归算法
  BiThrTree p = T->lchild; // p 指向根结点
  while(p!= T) // p 未指向头结点(遍历的最后 1 个结点的后继指向头结点)
  { Visit(p->data); // 访问根结点
    if(p->LTag == Link) // p 有左孩子
      p = p->lchild; // p 指向左孩子(后继)
    else // p 无左孩子
      p = p->rchild; // p 指向右孩子或后继
  }
}
void PostThreading(BiThrTree p)
{ // PostOrderThreading()调用的递归函数
  if(p) // p 不空
  { PostThreading(p->lchild); // 对 p 的左孩子递归调用 PostThreading()
    PostThreading(p->rchild); // 对 p 的右孩子递归调用 PostThreading()
    if(!p->lchild) // p 没有左孩子
    { p->LTag = Thread; // p 的左孩子指针为线索
      p->lchild = pre; // p 的左孩子指针指向前驱
    }
    if(!pre->rchild) // p 的前驱没有右孩子
    { pre->RTag = Thread; // p 前驱的右孩子指针为线索
      pre->rchild = p; // p 前驱的后继指向 p
    }
    pre = p; // 移动前驱
  }
}
void PostOrderThreading(BiThrTree &Thrt,BiThrTree T)
{ // 后序递归线索化二叉树
  if(!(Thrt = (BiThrTree)malloc(sizeof(BiThrNode)))) // 生成头结点
    exit(OVERFLOW);
  Thrt->LTag = Link; // 头结点的左孩子指针为孩子
  Thrt->RTag = Thread; // 头结点的右孩子指针为线索
  if(!T) // 空树
    Thrt->lchild = Thrt->rchild = Thrt; // 头结点的左右孩子指针指向自身
  else // 非空树
  { Thrt->lchild = Thrt->rchild = T; // 头结点的左右孩子指针指向根结点(最后一个结点)
    pre = Thrt; // 前驱为头结点
    PostThreading(T); // 从头结点开始后序递归线索化
    if(pre->RTag!= Link) // 最后一个结点没有右孩子
    { pre->RTag = Thread; // 最后一个结点的右孩子指针为线索
      pre->rchild = Thrt; // 最后一个结点的后继指向头结点
    }
  }
}
```

```
void DestroyBiTree(BiThrTree &T)
{ // DestroyBiThrTree 调用的递归函数,T 指向根结点
  if(T) // 非空树
  { if(T->LTag == 0) // 有左孩子
      DestroyBiTree(T->lchild); // 销毁左子树
    if(T->RTag == 0) // 有右孩子
      DestroyBiTree(T->rchild); // 销毁右子树
    free(T); // 释放根结点
    T = NULL; // 空指针赋 0
  }
}
void DestroyBiThrTree(BiThrTree &Thrt)
{ // 初始条件:线索二叉树 Thrt 存在。操作结果:销毁线索二叉树 Thrt
  if(Thrt) // 头结点存在
  { if(Thrt->lchild) // 根结点存在
      DestroyBiTree(Thrt->lchild); // 递归销毁头结点 lchild 所指二叉树
    free(Thrt); // 释放头结点
    Thrt = NULL; // 线索二叉树 Thrt 指针赋 0
  }
}
```

```
// main6-3.cpp 检验线索二叉树部分基本操作的程序
#define CHAR 1 // 字符型。第 2 行
//#define CHAR 0 // 整型(二者选一)。第 3 行
#include"func6-1.cpp" // 利用条件编译,在主程序中选择结点的类型,访问树结点的函数
#include"c6-3.h" // 二叉树的二叉线索存储结构
#include"bo6-4.cpp" // 构造线索二叉树的基本操作
#define FLAG 0 // 是否进行后序遍历的标志
void main()
{
  BiThrTree H,T;
  int i;
  for(i = 1;i<= 3;i++ ) // 循环 3 次,实现 3 种线索化和遍历
  {
#if CHAR // CHAR 值为非零,结点类型为字符
    printf("请按先序输入二叉树(如:ab 三个空格表示 a 为根结点,b 为左子树的二叉树):\n");
#else // CHAR 值为零,结点类型为整型
    printf("请按先序输入二叉树(如:1 2 0 0 0 表示 1 为根结点,2 为左子树的二叉树):\n");
#endif
    CreateBiThrTree(T); // 按先序产生二叉树
    scanf("%*c"); // 吃掉回车符
    switch(i)
    { case 1:PreOrderThreading(H,T); // 在先序遍历的过程中,先序线索化二叉树
             printf("先序遍历(输出)线索二叉树:\n");
```

```
                PreOrderTraverse_Thr(H,visit); // 先序遍历(输出)线索二叉树
                break;
        case 2:InOrderThreading(H,T); // 在中序遍历的过程中,中序线索化二叉树
                printf("中序遍历(输出)线索二叉树:\n");
                InOrderTraverse_Thr(H,visit); // 中序遍历(输出)线索二叉树
                break;
        case 3:PostOrderThreading(H,T); // 在后序遍历的过程中,后序线索化二叉树
#if FLAG // 当定义 FLAG 为 1 时,进行后序遍历
                printf("后序遍历(输出)线索二叉树:\n");
                PostOrderTraverse_Thr(H,visit);
#endif
      }
      printf("\n");
      DestroyBiThrTree(H); // 销毁线索二叉树 H
    }
}
```

**程序运行结果:**(输入中的␣表示空格)

```
请按先序输入二叉树(如:ab 三个空格表示 a 为根结点,b 为左子树的二叉树):
abdg␣␣␣e␣␣c␣f␣␣↙(见图 6-21)
先序遍历(输出)线索二叉树:(见图 6-22)
a b d g e c f
请按先序输入二叉树(如:ab 三个空格表示 a 为根结点,b 为左子树的二叉树):
abdg␣␣␣e␣␣c␣f␣␣↙(见图 6-21)
中序遍历(输出)线索二叉树:(见图 6-23)
g d b e a c f
请按先序输入二叉树(如:ab 三个空格表示 a 为根结点,b 为左子树的二叉树):
abdg␣␣␣e␣␣c␣f␣␣↙(见图 6-21)
```

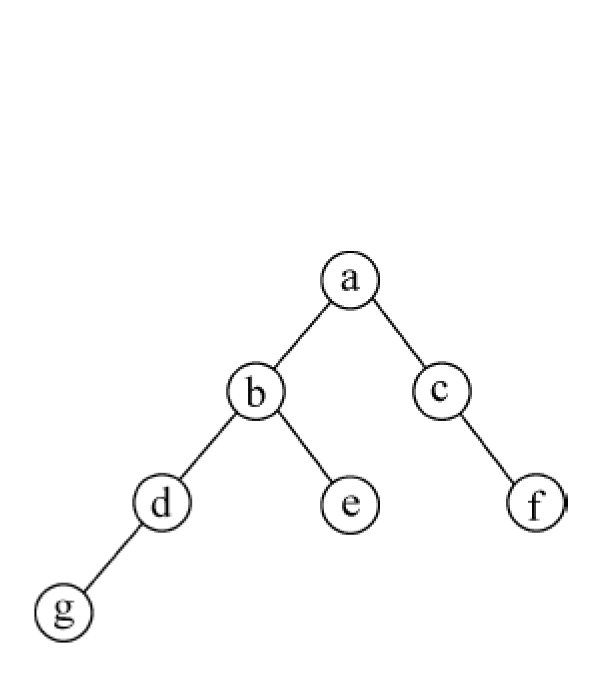


图 6-21　生成的二叉树

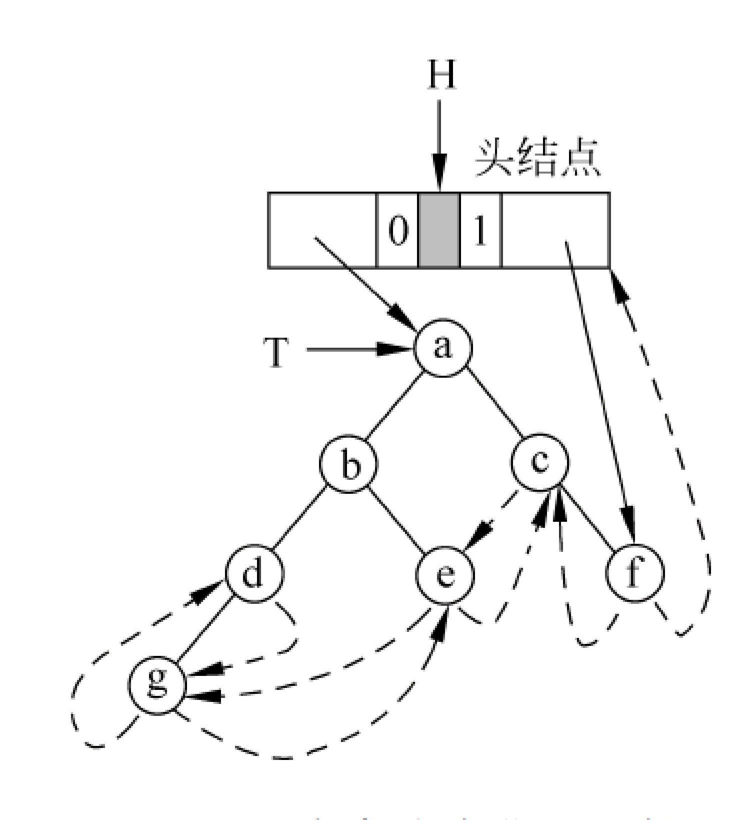


图 6-22　先序线索化二叉树

图 6-24 是后序线索化的示意图。由于没有指向双亲的指针,这种结构还是无法仅仅根据后序线索二叉树进行后序遍历。

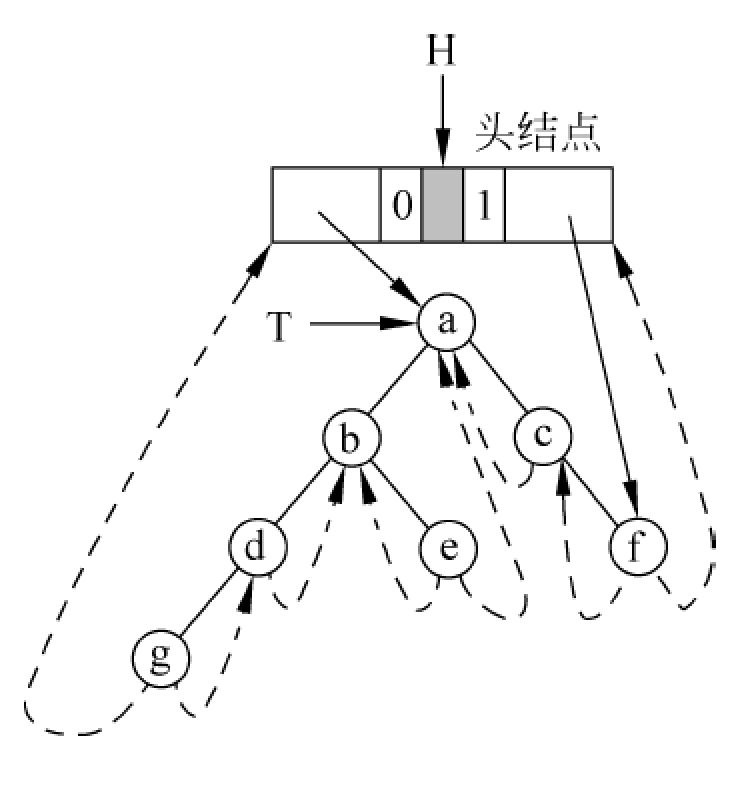



图 6-23　中序线索化二叉树

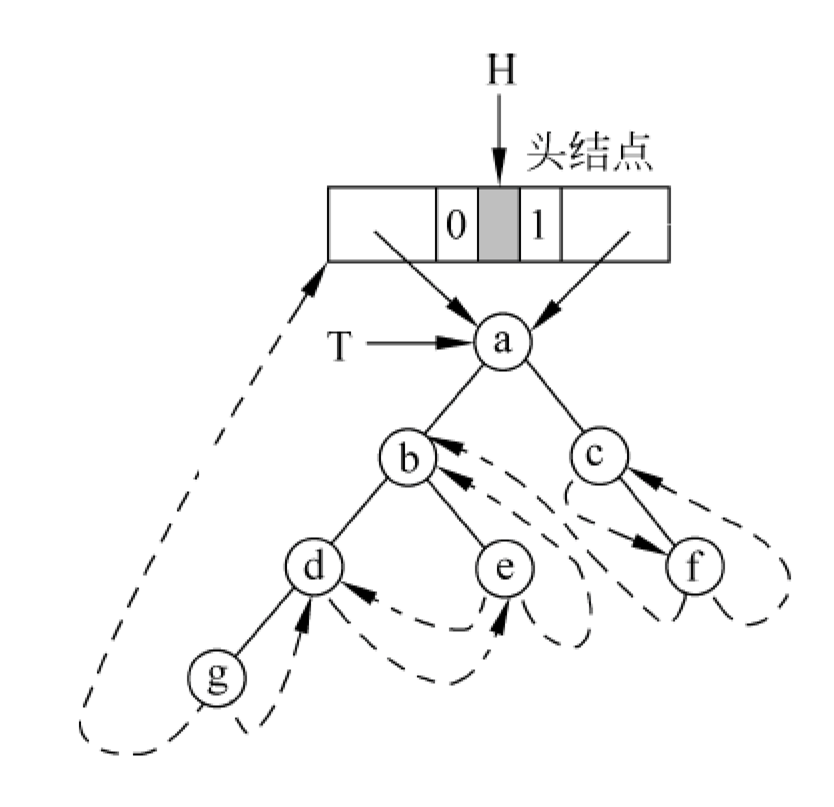



图 6-24　后序线索化二叉树

## 6.3　树 和 森 林

二叉树是最简单的树，还有多叉树，多叉树统称为树。森林是由 2 棵以上的树组成的。

```
// c6-4.h 树的二叉链表(孩子-兄弟)存储结构。在教科书第 136 页(见图 6-25)
typedef struct CSNode // 结点类型
{ TElemType data; // 结点的值
  CSNode * firstchild, * nextsibling;
}CSNode, * CSTree;
```

CSTree　CSNode

firstchild　data　nextsibling

CSNode　CSNode

图 6-25　树的二叉链表(孩子-兄弟)存储结构

一棵树无论有多少叉，它最多有一个长子和一个排序恰在其下的兄弟。根据这样的定义，则每个结点的结构就都统一到了二叉链表结构上。这样有利于对结点进行操作。图 6-26 是教科书图 6.13 所示的树的二叉链表(孩子-兄弟)存储结构。

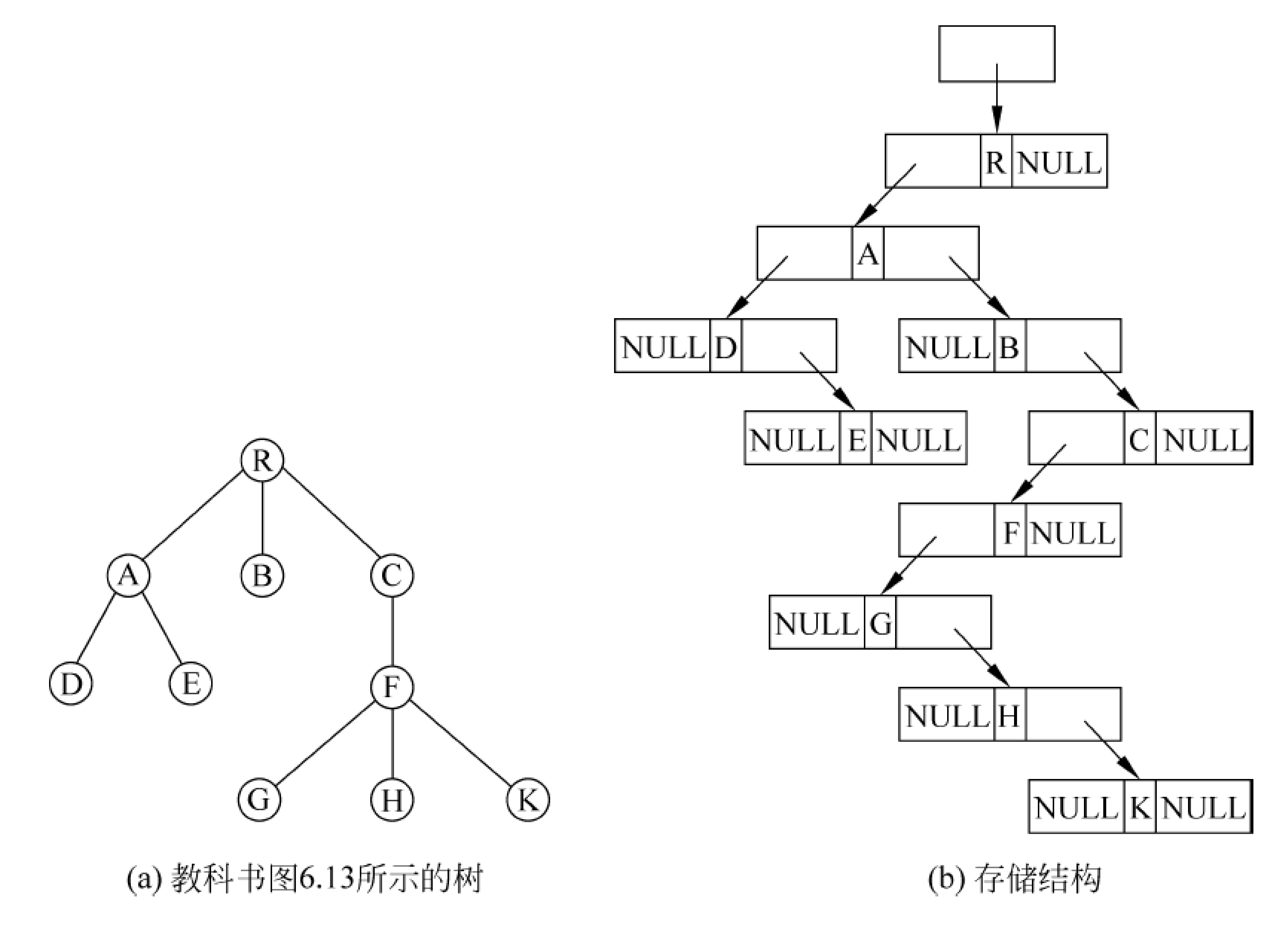



图 6-26　采用二叉链表(孩子-兄弟)存储树的示例

```
// bo6-5.cpp 树的二叉链表(孩子-兄弟)存储(存储结构由 c6-4.h 定义)的基本操作(16 个)
#define ClearTree DestroyTree // 二者操作相同
void InitTree(CSTree &T)
{ // 操作结果:构造空树 T
  T = NULL;
}
void DestroyTree(CSTree &T)
{ // 初始条件:树 T 存在。操作结果:销毁树 T
  if(T) // 非空树
  { DestroyTree(T->firstchild); // 递归销毁 T 的长子为根结点的子树
    DestroyTree(T->nextsibling); // 递归销毁 T 的下一个兄弟为根结点的子树
    free(T); // 释放根结点
    T = NULL; // 空指针赋 0
  }
}
typedef CSTree QElemType; // 定义队列元素类型为孩子-兄弟二叉链表的指针类型
#include"c3-2.h" // 定义 LinkQueue 类型(链队列)
#include"bo3-2.cpp" // LinkQueue 类型的基本操作
void CreateTree(CSTree &T)
{ // 构造树 T
  char c[20]; // 临时存放孩子结点(设不超过 20 个)的值
  CSTree p,p1;
  LinkQueue q;
  int i,m;
  InitQueue(q); // 初始化队列 q
  printf("请输入根结点(字符型,空格为空):");
  scanf("%c%*c",&c[0]); // 输入根结点的值
  if(c[0]!= Nil) // 非空树
  { T = (CSTree)malloc(sizeof(CSNode)); // 建立根结点
    T->data = c[0]; // 给根结点赋值
    T->nextsibling = NULL; // 根结点没有下一个兄弟
    EnQueue(q,T); // 入队根结点的指针
    while(!QueueEmpty(q)) // 队不空
    { DeQueue(q,p); // 出队一个结点的指针
      printf("请按长幼顺序输入结点%c的所有孩子:",p->data);
      gets(c); // 将结点的所有孩子作为字符串输入
      m = strlen(c);
      if(m>0) // 有孩子
      { p1 = p->firstchild = (CSTree)malloc(sizeof(CSNode)); // 建立长子结点
        p1->data = c[0]; // 给长子结点赋值
        EnQueue(q,p1); // 入队 p1 结点的指针
        for(i = 1;i<m;i++) // 对于除长子之外的其他孩子
        { p1->nextsibling = (CSTree)malloc(sizeof(CSNode)); // 建立下一个兄弟结点
          p1 = p1->nextsibling; // p1 指向下一个兄弟结点
          p1->data = c[i]; // 给 p1 所指结点赋值
          EnQueue(q,p1); // 入队 p1 结点的指针
```

```
            }
            p1->nextsibling = NULL; // 最后一个结点没有下一个兄弟
          }
          else // 无孩子
            p->firstchild = NULL; // 长子指针为空
        }
      }
      else
        T = NULL; // 空树
    }
    Status TreeEmpty(CSTree T)
    { // 初始条件：树 T 存在。操作结果：若 T 为空树，则返回 TURE；否则返回 FALSE
      if(T) // T 不空
        return FALSE;
      else
        return TRUE;
    }
    int TreeDepth(CSTree T)
    { // 初始条件：树 T 存在。操作结果：返回 T 的深度
      CSTree p;
      int depth,max = 0;
      if(!T) // 树空
        return 0;
      for(p = T->firstchild;p;p = p->nextsibling) // 对于树 T 根结点的所有孩子结点(由 p 指向)
      { // 求子树深度的最大值
        depth = TreeDepth(p); // 递归求孩子结点的深度 depth
        if(depth>max) // 最大孩子结点的深度存 max
          max = depth;
      }
      return max + 1; // 树的深度 = 子树深度最大值 + 1
    }
    TElemType Value(CSTree p)
    { // 返回 p 所指结点的值
      return p->data;
    }
    TElemType Root(CSTree T)
    { // 初始条件：树 T 存在。操作结果：返回 T 的根
      if(T)
        return Value(T);
      else
        return Nil;
    }
    CSTree Point(CSTree T,TElemType s)
    { // 返回二叉链表(孩子-兄弟)树 T 中指向元素值为 s 的结点的指针。新增
      LinkQueue q;
```

```
  QElemType a;
  if(T) // 非空树
  { InitQueue(q); // 初始化队列
    EnQueue(q,T); // 根结点入队
    while(!QueueEmpty(q)) // 队不空
    { DeQueue(q,a); // 出队,队列元素赋给 a
      if(a->data == s) // 找到元素值为 s 的结点
        return a; // 返回指向其的指针
      if(a->firstchild) // 有长子
        EnQueue(q,a->firstchild); // 入队长子
      if(a->nextsibling) // 有下一个兄弟
        EnQueue(q,a->nextsibling); // 入队下一个兄弟
    }
  }
  return NULL;
}
Status Assign(CSTree &T,TElemType cur_e,TElemType value)
{ // 初始条件:树 T 存在,cur_e 是树 T 中结点的值。操作结果:改 cur_e 为 value
  CSTree p;
  if(T) // 非空树
  { p = Point(T,cur_e); // p 为 cur_e 的指针
    if(p) // 找到 cur_e
    { p->data = value; // 赋新值
      return OK;
    }
  }
  return ERROR; // 树空或未找到
}
TElemType Parent(CSTree T,TElemType cur_e)
{ // 初始条件:树 T 存在,cur_e 是 T 中某个结点
  // 操作结果:若 cur_e 是 T 的非根结点,则返回它的双亲;否则函数值为“空”
  CSTree p,t;
  LinkQueue q;
  InitQueue(q); // 初始化队列 q
  if(T) // 树非空
  { if(Value(T) == cur_e) // 根结点值为 cur_e
      return Nil;
    EnQueue(q,T); // 根结点入队
    while(!QueueEmpty(q)) // 队列不空
    { DeQueue(q,p); // 出队元素(指针)赋给 p
      if(p->firstchild) // p 有长子
      { if(p->firstchild->data == cur_e) // 长子为 cur_e
          return Value(p); // 返回双亲的值
        t = p; // 双亲指针赋给 t
        p = p->firstchild; // p 指向长子
```

```
          EnQueue(q,p); // 入队长子
          while(p->nextsibling) // 有下一个兄弟
          { p=p->nextsibling; // p 指向下一个兄弟
            if(Value(p)==cur_e) // 下一个兄弟为 cur_e
              return Value(t); // 返回双亲的值
            EnQueue(q,p); // 入队下一个兄弟
          }
        }
      }
    }
  return Nil; // 树空或未找到 cur_e
}
TElemType LeftChild(CSTree T,TElemType cur_e)
{ // 初始条件：树 T 存在，cur_e 是 T 中某个结点
  // 操作结果：若 cur_e 是 T 的非叶子结点，则返回它的最左孩子；否则返回"空"
  CSTree f;
  f=Point(T,cur_e); // f 指向结点 cur_e
  if(f&&f->firstchild) // 找到结点 cur_e 且结点 cur_e 有长子
    return f->firstchild->data; // 返回结点 cur_e 的长子的值
  else
    return Nil; // 返回空
}
TElemType RightSibling(CSTree T,TElemType cur_e)
{ // 初始条件：树 T 存在，cur_e 是 T 中某个结点
  // 操作结果：若 cur_e 有右兄弟，则返回它的右兄弟；否则返回"空"
  CSTree f;
  f=Point(T,cur_e); // f 指向结点 cur_e
  if(f&&f->nextsibling) // 找到结点 cur_e 且结点 cur_e 有右兄弟
    return f->nextsibling->data; // 返回结点 cur_e 的右兄弟的值
  else
    return Nil; // 返回空
}
Status InsertChild(CSTree &T,CSTree p,int i,CSTree c)
{ // 初始条件：树 T 存在，p 指向 T 中某个结点，1≤i≤p 所指结点的度+1，非空树 c 与 T 不相交
  // 操作结果：插入 c 为 T 中 p 结点的第 i 棵子树。因为 p 所指结点的地址不会改变，
  //           故 p 不需要是引用类型
  int j;
  CSTree q;
  if(T) // T 不空
  { if(i==1) // 插入 c 为 p 的长子
    { c->nextsibling=p->firstchild; // p 的原长子现是 c 的下一个兄弟(c 本无兄弟)
      p->firstchild=c; // p 的长子指针指向 c(c 成为 p 的长子)
    }
    else // c 不是 p 的长子
    { q=p->firstchild; // q 指向 p 的长子结点
```

```
      j = 2;
      while(q&&j<i) // 找 c 的插入点,并由 q 指向
      { q = q->nextsibling; // q 指向下一个兄弟结点
        j++; // 计数 + 1
      }
      if(j == i) // 找到插入位置
      { c->nextsibling = q->nextsibling; // c 的下一个兄弟指向 p 的原第 i 个孩子
        q->nextsibling = c; // 在 p 中插入 c 作为 p 的第 i 个孩子
      }
      else // p 原有孩子数小于 i - 1
        return ERROR;
    }
    return OK;
  }
  else // T 空
    return ERROR;
}
Status DeleteChild(CSTree &T,CSTree p,int i)
{ // 初始条件:树 T 存在,p 指向 T 中某个结点,1≤i≤p 所指结点的度
  // 操作结果:删除 T 中 p 所指结点的第 i 棵子树。
  //           因为 p 所指结点的地址不会改变,故 p 不需要是引用类型
  CSTree b,q;
  int j;
  if(T) // T 不空
  { if(i == 1) // 删除长子
    { // 把长子结点子树从 p 中分离出来
      b = p->firstchild; // b 指向 p 的长子结点
      p->firstchild = b->nextsibling; // p 的原次子现是长子
      b->nextsibling = NULL; // p 的长子结点成为待删除子树的根结点,其下一个兄弟指针为空
      DestroyTree(b); // 销毁由 b 指向的 p 的长子结点子树
    }
    else // 删除非长子
    { q = p->firstchild; // q 指向 p 的长子结点
      j = 2;
      while(q&&j<i) // 找第 i 棵子树
      { q = q->nextsibling; // q 指向下一个兄弟结点
        j++; // 计数 + 1
      }
      if(j == i) // 找到第 i 棵子树
      { b = q->nextsibling; // b 指向待删除子树
        q->nextsibling = b->nextsibling; // 从树 p 中删除这棵子树
        b->nextsibling = NULL; // 待删除子树的根结点的下一个兄弟指针为空
        DestroyTree(b); // 销毁由 b 指向的 p 的长子结点子树
      }
```

```
        else // p原有孩子数小于i
          return ERROR;
      }
      return OK;
    }
    else
      return ERROR;
}
void PostOrderTraverse(CSTree T,void(* Visit)(TElemType))
{ // 后根遍历孩子-兄弟二叉链表结构的树T
  CSTree p;
  if(T)
  { if(T->firstchild) // 有长子
    { PostOrderTraverse(T->firstchild,Visit); // 后根遍历长子子树
      p = T->firstchild->nextsibling; // p指向长子的下一个兄弟
      while(p) // 还有下一个兄弟
      { PostOrderTraverse(p,Visit); // 后根遍历下一个兄弟子树
        p = p->nextsibling; // p指向再下一个兄弟
      }
    }
    Visit(Value(T)); // 最后访问根结点
  }
}
void LevelOrderTraverse(CSTree T,void(* Visit)(TElemType))
{ // 层序遍历孩子-兄弟二叉链表结构的树T
  CSTree p;
  LinkQueue q;
  InitQueue(q); // 初始化队列q
  if(T) // 树非空
  { Visit(Value(T)); // 先访问根结点
    EnQueue(q,T); // 入队根结点的指针
    while(!QueueEmpty(q)) // 队不空
    { DeQueue(q,p); // 出队一个结点的指针
      if(p->firstchild) // 有长子
      { p = p->firstchild; // p指向长子结点
        Visit(Value(p)); // 访问长子结点
        EnQueue(q,p); // 入队长子结点的指针
        while(p->nextsibling) // 有下一个兄弟
        { p = p->nextsibling; // p指向下一个兄弟结点
          Visit(Value(p)); // 访问下一个兄弟
          EnQueue(q,p); // 入队兄弟结点的指针
        }
      }
    }
  }
```

```
  printf("\n");
}

// bo6-6.cpp main6-4.cpp 和 algo7-3.cpp 调用
void PreOrderTraverse(CSTree T,void(* Visit)(TElemType))
{ // 先根遍历孩子-兄弟二叉链表结构的树 T
  if(T)
  { Visit(T->data); // 先访问根结点
    PreOrderTraverse(T->firstchild,Visit); // 再先根遍历长子子树
    PreOrderTraverse(T->nextsibling,Visit); // 最后先根遍历下一个兄弟子树
  }
}

// main6-4.cpp 检验 bo6-5.cpp 和 bo6-6.cpp 的程序
#define CHAR 1 // 只可是字符型
#include"func6-1.cpp" // 利用条件编译,在主程序中选择结点的类型,访问树结点的函数
#include"c6-4.h" // 树的二叉链表(孩子-兄弟)存储结构
#include"bo6-5.cpp" // 树的二叉链表(孩子-兄弟)基本操作
#include"bo6-6.cpp" // 包括 PreOrderTraverse()
void main()
{
  int i;
  CSTree T,p,q;
  TElemType e,e1;
  InitTree(T); // 构造空树 T
  printf("构造空树后,树空否?%d(1:是 0:否)。树根为%c,树的深度为%d。\n",
  TreeEmpty(T),Root(T),TreeDepth(T));
  CreateTree(T); // 按层序构造树 T
  printf("构造树 T 后,树空否?%d(1:是 0:否)。树根为%c,树的深度为%d。\n",
  TreeEmpty(T),Root(T),TreeDepth(T));
  printf("层序遍历树 T:\n");
  LevelOrderTraverse(T,visit); // 层序遍历树 T
  printf("请输入待修改的结点的值 新值:");
  scanf("%c%*c%c%*c",&e,&e1);
  Assign(T,e,e1); // 将树 T 中结点值为 e 的修改为 e1
  printf("层序遍历修改后的树 T:\n");
  LevelOrderTraverse(T,visit); // 层序遍历树 T
  printf("%c 的双亲是%c,长子是%c,下一个兄弟是%c。\n",e1,Parent(T,e1),
  LeftChild(T,e1),RightSibling(T,e1));
  printf("建立树 p:\n");
  CreateTree(p); // 按层序构造树 p
  printf("层序遍历树 p:\n");
  LevelOrderTraverse(p,visit); // 层序遍历树 p
  printf("将树 p 插到树 T 中,请输入 T 中 p 的双亲结点 子树序号:");
  scanf("%c%d%*c",&e,&i);
```

```
    q = Point(T,e); // 将指向树 T 中结点 e 的指针赋给 q
    InsertChild(T,q,i,p); // 将树 p 插入树 T 中作为 q 所指结点的第 i 棵子树
    printf("层序遍历修改后的树 T:\n");
    LevelOrderTraverse(T,visit); // 层序遍历树 T
    printf("先根遍历树 T: \n");
    PreOrderTraverse(T,visit); // 先根遍历树 T
    printf("\n 后根遍历树 T: \n");
    PostOrderTraverse(T,visit); // 后根遍历树 T
    printf("\n 删除树 T 中结点 e 的第 i 棵子树,请输入 e i: ");
    scanf("%c%d",&e,&i);
    q = Point(T,e); // 将指向树 T 中结点 e 的指针赋给 q
    DeleteChild(T,q,i); // 删除树 T 中 q 所指结点的第 i 棵子树
    printf("层序遍历修改后的树 T:\n");
    LevelOrderTraverse(T,visit); // 层序遍历树 T
    DestroyTree(T); // 销毁树 T
}
```

**程序运行结果:**

```
构造空树后,树空否? 1(1: 是 0: 否)。树根为 ,树的深度为 0。
请输入根结点(字符型,空格为空): R↙
请按长幼顺序输入结点 R 的所有孩子: ABC↙
请按长幼顺序输入结点 A 的所有孩子: DE↙
请按长幼顺序输入结点 B 的所有孩子: ↙
请按长幼顺序输入结点 C 的所有孩子: F↙
请按长幼顺序输入结点 D 的所有孩子: ↙
请按长幼顺序输入结点 E 的所有孩子: ↙
请按长幼顺序输入结点 F 的所有孩子: GHK↙
请按长幼顺序输入结点 G 的所有孩子: ↙
请按长幼顺序输入结点 H 的所有孩子: ↙
请按长幼顺序输入结点 K 的所有孩子: ↙
构造树 T 后,树空否? 0(1: 是 0: 否)。树根为 R,树的深度为 4。
层序遍历树 T:(见图 6-26(a))
R A B C D E F G H K
请输入待修改的结点的值 新值: D d↙
层序遍历修改后的树 T:
R A B C d E F G H K
d 的双亲是 A,长子是 ,下一个兄弟是 E。
建立树 p:
请输入根结点(字符型,空格为空): f↙
请按长幼顺序输入结点 f 的所有孩子: ghk↙
请按长幼顺序输入结点 g 的所有孩子: ↙
请按长幼顺序输入结点 h 的所有孩子: ↙
请按长幼顺序输入结点 k 的所有孩子: ↙
层序遍历树 p:(见图 6-27)
f g h k
```



图 6-27　树 p 图示

将树 p 插到树 T 中，请输入 T 中 p 的双亲结点 子树序号：R 3↙

层序遍历修改后的树 T：（见图 6-28）

R A B f C d E g h k F G H K

先根遍历树 T：

R A d E B f g h k C F G H K

后根遍历树 T：

d E A B g h k f G H K F C R

删除树 T 中结点 e 的第 i 棵子树，请输入 e i：C 1↙（见图 6-29）

层序遍历修改后的树 T：

R A B f C d E g h k

图 6-28　树 p 插到树 T 后的状况

图 6-29　删除树 T 中结点 C 的第 1 棵子树后的状况

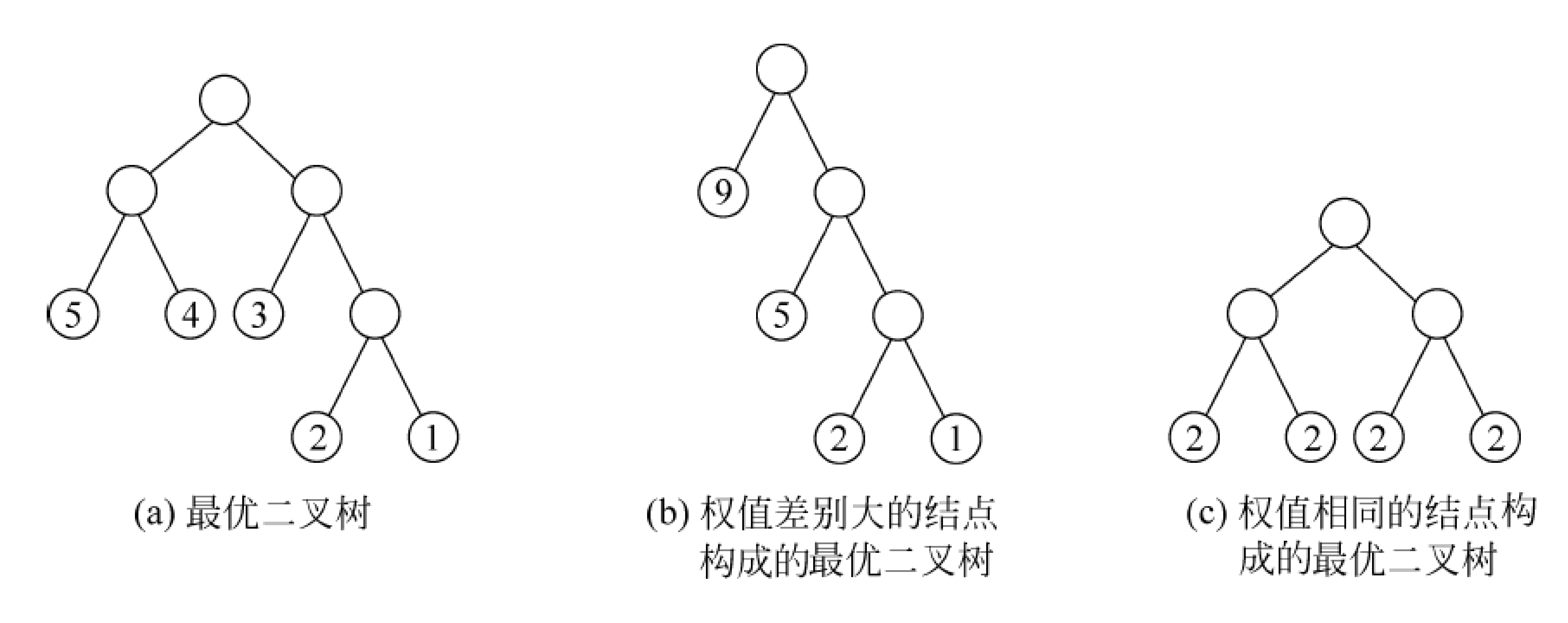
# 6.4　赫夫曼树及其应用

## 6.4.1　最优二叉树(赫夫曼树)

最优二叉树是带权路径长度最短的二叉树。根据结点的个数、权值的不同，最优二叉树的形状也各不相同。图 6-30 是 3 棵最优二叉树的例子。它们的共同特点是：带权值的结点都是叶子结点。权值越小的结点，其到根结点的路径越长。构造最优二叉树的方法如下：

（1）将每个带有权值的结点作为一棵仅有根结点的二叉树，树的权值为结点的权值；

（2）将其中两棵权值最小的树组成一棵新二叉树，新树的权值为两棵树的权值之和；




(a) 最优二叉树　(b) 权值差别大的结点构成的最优二叉树　(c) 权值相同的结点构成的最优二叉树

图 6-30　3 棵形状不同的最优二叉树

(3) 重复(2),直到所有结点都在一棵二叉树上,这棵二叉树就是最优二叉树。

最优二叉树的左右子树是可以互换的,因为这不影响树的带权路径长度。当结点的权值差别大到一定程度,最优二叉树就形成了如图 6-30(b)所示的"一边倒"的形状。当所有结点的权值一样,或其权值差别很小,最优二叉树就形成了如图 6-30(c)所示的完全二叉树的形状。叶子结点的路径长度近似相等。

最优二叉树除了叶子结点就是度为 2 的结点,没有度为 1 的结点。这样才使得树的带权路径长度最短。根据二叉树的性质 3,最优二叉树的结点数为叶子数的 2 倍减 1。

### 6.4.2 赫夫曼编码

```
// c6-5.h 赫夫曼树和赫夫曼编码的存储结构(见图 6-31)
typedef struct // 结点的结构,在教科书第 147 页
{ unsigned int weight; // 结点的权值
  unsigned int parent,lchild,rchild;
}HTNode, * HuffmanTree; // 动态分配数组存储
                        // 赫夫曼树
typedef char ** HuffmanCode; // 动态分配数组存
                             // 储赫夫曼编码表
```



图 6-31 赫夫曼树和赫夫曼编码的存储结构

c6-5.h 定义的二叉树结构可以称为"静态三叉链表二叉树结构"。是在前面没有讨论过的,但它特别适合建立赫夫曼树。赫夫曼树是由多棵二叉树(森林)组合成而的一棵树。这种二叉树结构既适合表示树,也适合表示森林。赫夫曼树结点的结构包括权值、双亲及左右孩子静态指针,双亲值为 0 的是根结点,左右孩子值均为 0 的是叶子结点。这种二叉树结构是动态生成的顺序结构。当叶子结点数确定,赫夫曼树的结点数也就确定了。由图 6-32(d)可见,建成的赫夫曼树除 0 号结点空间不用外,每个结点空间都未空置。

```
// func6-2.cpp 程序 algo6-1.cpp 和 algo6-2.cpp 要调用的 2 个函数
#define Order // 定义 Order。第 2 行
int min(HuffmanTree t,int i)
{ // 返回赫夫曼树 t 的前 i 个结点中权值最小的树的根结点序号,函数 select()调用
  int j,m;
  unsigned int k = UINT_MAX; // k 存最小权值,初值取为不小于可能的值(无符号整型最大值)
  for(j = 1;j<= i;j++ ) // 对于前 i 个结点
    if(t[j].weight<k&&t[j].parent == 0) // t[j]的权值小于 k,又是树的根结点
    { k = t[j].weight; // t[j]的权值赋给 k
      m = j; // 序号赋给 m
    }
  t[m].parent = 1; // 给选中的根结点的双亲赋非零值,避免第 2 次查找该结点
  return m; // 返回权值最小的根结点的序号
}
void select(HuffmanTree t,int i,int &s1,int &s2)
{ // 在赫夫曼树 t 的前 i 个结点中选择 2 个权值最小的树的根结点序号,s1 为其中序号(权值)较小的
#ifdef Order // 如果在主程中定义了 Order,则以下语句起作用
```

```
  int j;
#endif
  s1 = min(t,i); // 权值最小的根结点序号
  s2 = min(t,i); // 权值第 2 小的根结点序号
#ifdef Order // 如果在主程中定义了 Order,则执行下面一段程序
  if(s1>s2) // s1 的序号大于 s2 的
  { // 交换
    j = s1;
    s1 = s2; // s1 是权值最小的 2 个中序号较小的
    s2 = j; // s2 是权值最小的 2 个中序号较小的
  }
#endif
}
```

```
// func6-3.cpp 算法 6.12 的前半部分
  int m,i,s1,s2;
  unsigned c;
  HuffmanTree p;
  char *cd;
  if(n<=1) // 叶子结点数不大于 n
    return;
  m = 2*n-1; // n 个叶子结点的赫夫曼树共有 m 个结点
  HT = (HuffmanTree)malloc((m+1)*sizeof(HTNode)); // 0 号单元未用
  for(p = HT+1,i = 1;i<= n;++i,++p,++w) // 从 1 号单元开始到 n 号单元,给叶子结点赋值
  { // p 的初值指向 1 号单元
    (*p).weight = *w; // 赋权值
    (*p).parent = 0; // 双亲域为空(是根结点)
    (*p).lchild = 0; // 左右孩子为空(是叶子结点,即单结点树)
    (*p).rchild = 0;
  }
  for(;i<= m;++i,++p) // i 从 n+1 到 m
    (*p).parent = 0; // 其余结点的双亲域初值为 0
  for(i = n+1;i<= m;++i) // 建赫夫曼树
  { // 在 HT[1～i-1]中选择 parent 为 0 且 weight 最小的两个结点,其序号分别为 s1 和 s2
    select(HT,i-1,s1,s2);
    HT[s1].parent = HT[s2].parent = i; // i 号单元是 s1 和 s2 的双亲
    HT[i].lchild = s1; // i 号单元的左右孩子分别是 s1 和 s2
    HT[i].rchild = s2;
    HT[i].weight = HT[s1].weight + HT[s2].weight; // i 号单元的权值是 s1 和 s2 的权值之和
  }
```

```
// func6-4.cpp 求赫夫曼编码的主函数
void main()
{
```

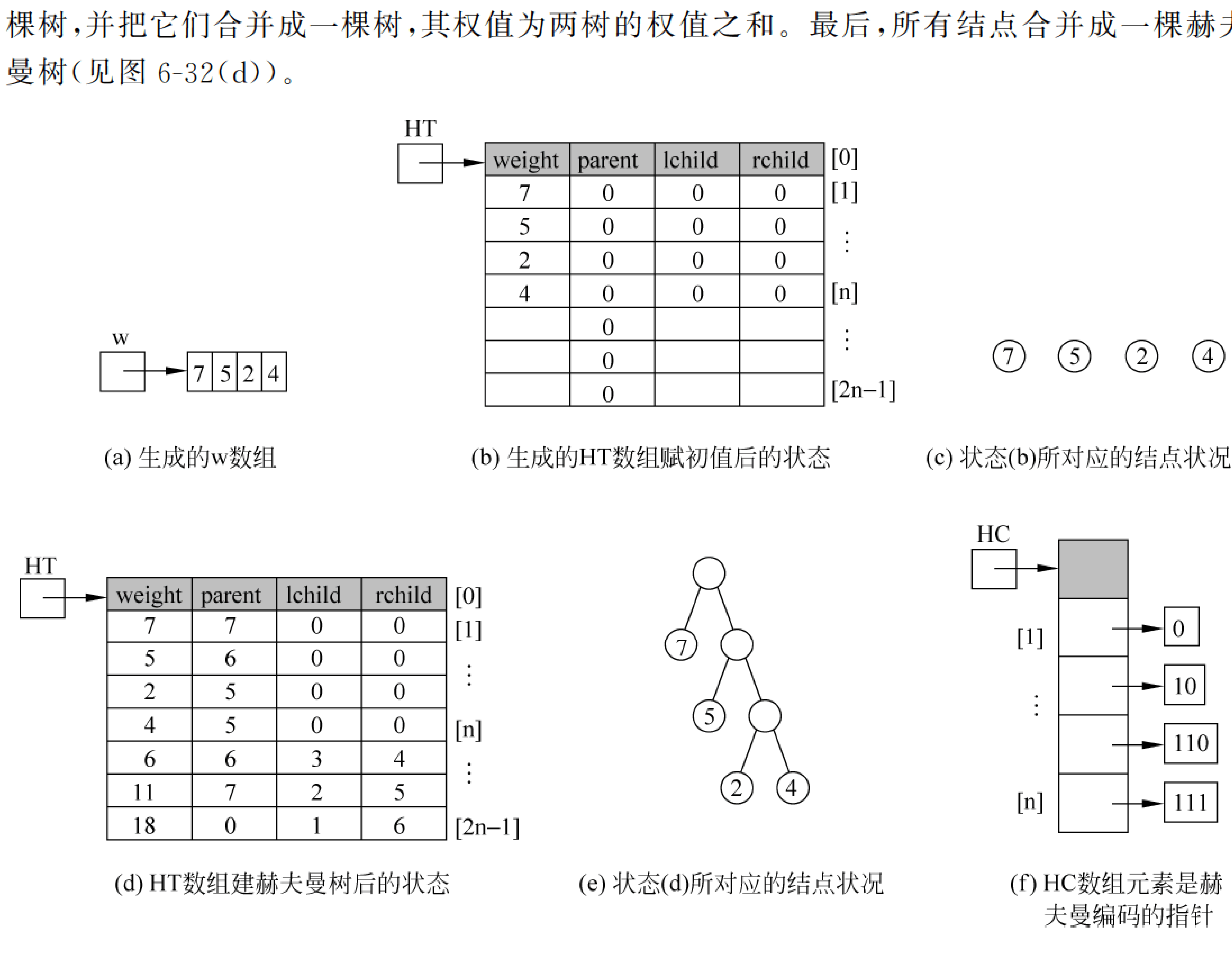
```
    HuffmanTree HT;
    HuffmanCode HC;
    int *w,n,i;
    printf("请输入权值的个数(>1):");
    scanf("%d",&n);
    w=(int*)malloc(n*sizeof(int)); // 动态生成存放 n 个权值的空间
    printf("请依次输入 %d 个权值(整型):\n",n);
    for(i=0;i<=n-1;i++)
      scanf("%d",w+i); // 依次输入权值
    HuffmanCoding(HT,HC,w,n); // 根据 w 所存的 n 个权值构造赫夫曼树 HT,n 个赫夫曼编码存于 HC
    for(i=1;i<=n;i++)
      puts(HC[i]); // 依次输出赫夫曼编码
  }
```

```
// algo6-1.cpp 求赫夫曼编码。实现算法 6.12 的程序
#include"c1.h"
#include"c6-5.h" // 赫夫曼树和赫夫曼编码的存储结构
#include"func6-2.cpp" // algo6-1.cpp 和 algo6-2.cpp 要调用的 2 个函数
void HuffmanCoding(HuffmanTree &HT,HuffmanCode &HC,int * w,int n) // 算法 6.12
{ // w 存放 n 个字符的权值(均>0),构造赫夫曼树 HT,并求出 n 个字符的赫夫曼编码 HC
  int start;
  unsigned f;
#include"func6-3.cpp" // 算法 6.12 的前半部分,以下是从叶子到根逆向求每个字符的赫夫曼编码
  HC=(HuffmanCode)malloc((n+1)*sizeof(char*));
  // 分配 n 个字符编码的头指针向量([0]不用)
  cd=(char*)malloc(n*sizeof(char)); // 分配求编码的工作空间
  cd[n-1]='\0'; // 编码结束符
  for(i=1;i<=n;i++)
  { // 逐个字符求赫夫曼编码
    start=n-1; // 编码结束符位置
    for(c=i,f=HT[i].parent;f!=0;c=f,f=HT[f].parent) // 从叶子到根逆向求编码
      if(HT[f].lchild==c) // c 是其双亲的左孩子
        cd[--start]='0'; // 由叶子向根赋值'0'
      else // c 是其双亲的右孩子
        cd[--start]='1'; // 由叶子向根赋值'1'
    HC[i]=(char*)malloc((n-start)*sizeof(char)); // 为第 i 个字符编码分配空间
    strcpy(HC[i],&cd[start]); // 从 cd 复制编码(串)到 HC
  }
  free(cd); // 释放工作空间
}
#include"func6-4.cpp" // 主函数
```

**程序运行结果**(以教科书图 6.24 为例,如图 6-32 所示):

```
请输入权值的个数(>1)：4↙
请依次输入 4 个权值(整型)：
7 5 2 4↙
0
10
110
111
```

图 6-32 是运行过程的图解。初始状态下(见图 6-32(b)),权值分别为 7、5、2、4 的 4 个结点是 4 棵独立的树(根结点)。它们没有双亲,也没有左右孩子。反复查找权值最小的两棵树,并把它们合并成一棵树,其权值为两树的权值之和。最后,所有结点合并成一棵赫夫曼树(见图 6-32(d))。



| weight | parent | lchild | rchild | |
|---|---|---|---|---|
| | | | | [0] |
| 7 | 0 | 0 | 0 | [1] |
| 5 | 0 | 0 | 0 | ⋮ |
| 2 | 0 | 0 | 0 | |
| 4 | 0 | 0 | 0 | [n] |
| | 0 | | | ⋮ |
| | 0 | | | |
| | 0 | | | [2n−1] |

| weight | parent | lchild | rchild | |
|---|---|---|---|---|
| | | | | [0] |
| 7 | 7 | 0 | 0 | [1] |
| 5 | 6 | 0 | 0 | ⋮ |
| 2 | 5 | 0 | 0 | |
| 4 | 5 | 0 | 0 | [n] |
| 6 | 6 | 3 | 4 | ⋮ |
| 11 | 7 | 2 | 5 | |
| 18 | 0 | 1 | 6 | [2n−1] |



(a) 生成的w数组　(b) 生成的HT数组赋初值后的状态　(c) 状态(b)所对应的结点状况

(d) HT数组建赫夫曼树后的状态　(e) 状态(d)所对应的结点状况　(f) HC数组元素是赫夫曼编码的指针

图 6-32　algo6-1.cpp 运行过程的图解

分析教科书中图 6.26 及图 6.27 可见,在找到两个权值最小的树的根结点后,合并成一棵树时,是将序号小的结点作为左子树,序号大的结点作为右子树的。为了和教科书中的结果一致,在 select()函数(在 func6-2.cpp 中)中,增加了需要时交换序号的程序段。其实这是不必要的。在含光盘的教科书中有关赫夫曼树的课件又是将权值小的结点作为左子树,权值大的结点作为右子树的。这样做从算法实现上是更简单易行的。为了在同一个程序中能够演示这两种情况,在 func6-2.cpp 中增加了条件编译。删除 func6-2.cpp 中的第 2 行,则运行结果与光盘的课件一致;否则运行结果与教科书中图 6.26 及图 6.27 一致。这两种方法的赫夫曼编码的形式虽然不同。但每个权值的码长相同,都是赫夫曼编码。

algo6-2.cpp 与 algo6-1.cpp 仅是函数 HuffmanCoding()的后半部分不同,也就是在建

立好赫夫曼树后求赫夫曼编码的方法不同。algo6-1.cpp(算法 6.12)是依次从叶子结点(左右孩子均为 0 的结点,也就是序号在 1～n 之间的结点)开始,根据其双亲结点的序号,逐步找到根结点的。这种方法最先求得的是最后一位编码(叶子结点的编码),最后求得第 1 位编码。而 algo6-2.cpp(算法 6.13)求赫夫曼编码的方法是由根结点起,依次查找其左右孩子,直到找到叶子结点。它最先求得的是第 1 位编码。它是按照赫夫曼树的结构,从左到右依次求得每个叶子结点的赫夫曼编码的。在完成了一个叶子结点的编码,求下一个叶子结点的编码时,不用再从根结点开始,只是回溯到这两个叶子结点的“分叉处”,继续求编码即可。两个叶子结点在“分叉处”之前的编码是一样的。为了能够回溯到正确的位置,需要对每个结点标注其左右分支是否已被编码。由于建立好赫夫曼树后,weight 域不再起作用,故可用其做标注。

```
// algo6-2.cpp 实现算法 6.13 的程序
#include"c1.h"
#include"c6-5.h" // 赫夫曼树和赫夫曼编码的存储结构
#include"func6-2.cpp" // algo6-1.cpp 和 algo6-2.cpp 要调用的 2 个函数
void HuffmanCoding(HuffmanTree &HT,HuffmanCode &HC,int *w,int n)
// 前半部分为算法 6.12 的前半部分
{ // w 存放 n 个字符的权值(均>0),构造赫夫曼树 HT,并求出 n 个字符的赫夫曼编码 HC
  unsigned cdlen;
#include"func6-3.cpp" // 算法 6.12 的前半部分
  // 以下为算法 6.13,无栈非递归遍历赫夫曼树,求赫夫曼编码
  HC=(HuffmanCode)malloc((n+1)*sizeof(char*));
  // 分配 n 个字符编码的头指针向量([0]不用)
  cd=(char*)malloc(n*sizeof(char)); // 分配求编码的工作空间
  c=m; // m=2×n-1,从最后一个结点(根)开始
  cdlen=0; // 码长的初值为 0
  for(i=1;i<=m;++i)
    HT[i].weight=0; // 求编码不再需要权值域,改作结点状态标志,0 表示其左右孩子都不曾被访问
  while(c) // 未到叶子结点的孩子域
  { if(HT[c].weight==0) // 左右孩子都不曾被访问
    { // 向左
      HT[c].weight=1; // 左孩子被访问过,右孩子不曾被访问的标志
      if(HT[c].lchild!=0) // 有左孩子(不是叶子结点)
      { c=HT[c].lchild; // 置 c 为其左孩子序号(向叶子方向走一步)
        cd[cdlen++]='0'; // 左分支编码为 0
      }
      else if(HT[c].rchild==0) // 序号 c 为叶子结点(既没有左孩子,也没有右孩子)
      { // 登记叶子结点的字符的编码
        HC[c]=(char*)malloc((cdlen+1)*sizeof(char)); // 生成编码空间
        cd[cdlen]='\0'; // 最后一个位置赋 0(串结束符)
        strcpy(HC[c],cd); // 复制编码(串)
      }
    }
    else if(HT[c].weight==1) // 左孩子被访问过,右孩子不曾被访问
```

```
    { // 向右
      HT[c].weight = 2; // 左右孩子均被访问过的标志
      if(HT[c].rchild! = 0) // 有右孩子(不是叶子结点)
      { c = HT[c].rchild; // 置 c 为其右孩子序号(向叶子方向走一步)
        cd[cdlen++ ] = '1'; // 右分支编码为 1
      }
    }
    else // 左右孩子均被访问过(HT[c].weight == 2),向根结点方向退一步
    { c = HT[c].parent; // 置 c 为其双亲序号(向根方向退一步)
      -- cdlen; // 退到父结点,编码长度减 1
    }
  }
  free(cd); // 释放求编码的空间
}
#include"func6-4.cpp" // 主函数
```

**程序运行结果**(以教科书例 6-2 为例):

```
请输入权值的个数(>1):8↙
请依次输入 8 个权值(整型):
5 29 7 8 14 23 3 11↙
0110
10
1110
1111
110
00
0111
010
```

# 第 7 章

# 图

## 7.1 图的存储结构

图是比较复杂的数据结构,它由顶点和顶点之间的弧或边组成。任何两个顶点之间都可能存在弧或边。利用计算机存储图的信息,就要求能存储有关图的所有信息。也就是要存储图的顶点个数及每个顶点的特征、每对顶点之间是否存在弧(边)及弧(边)的特征。本章介绍了两种图的存储结构,它们各有特点,都能够存储图的所有信息。

### 7.1.1 数组表示法

```
// c7-1.h 图的数组(邻接矩阵)存储结构。在教科书第 161 页(见图 7-1)
#define INFINITY INT_MAX // 用整型最大值代替∞
typedef int VRType; // 定义顶点关系类型为整型,与 INFINITY 的类型一致
#define MAX_VERTEX_NUM 26 // 最大顶点个数
enum GraphKind{DG,DN,UDG,UDN}; // {有向图,有向网,无向图,无向网}
typedef struct // 边(弧)信息结构
{ VRType adj; // 顶点关系类型。对无权图,用 1(是)或 0(否)表示相邻否;对带权图,则为权值
  InfoType * info; // 该弧相关信息的指针(可无)
}ArcCell,AdjMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM]; // 二维数组
struct MGraph // 图的结构
{ VertexType vexs[MAX_VERTEX_NUM]; // 顶点向量
  AdjMatrix arcs; // 邻接矩阵(二维数组)
  int vexnum,arcnum; // 图的当前顶点数和弧数
  GraphKind kind; // 图的种类标志
};
```

在 c7-1.h 中,定义顶点关系类型 VRType 为整型,对于图,其值是 0 或 1;对于网(带权图),其值是权值,一般是整型,也可根据情况定义为浮点型。那样就需要把 INFINITY 也相应定义为浮点型最大值。

结构体 MGraph 中存储顶点信息的类型是 VertexType 类型(顶点类型),它一般是结构体,用来存储有关顶点的一切信息。如:顶点名称、顶点坐标等。图的顶点信息因图而

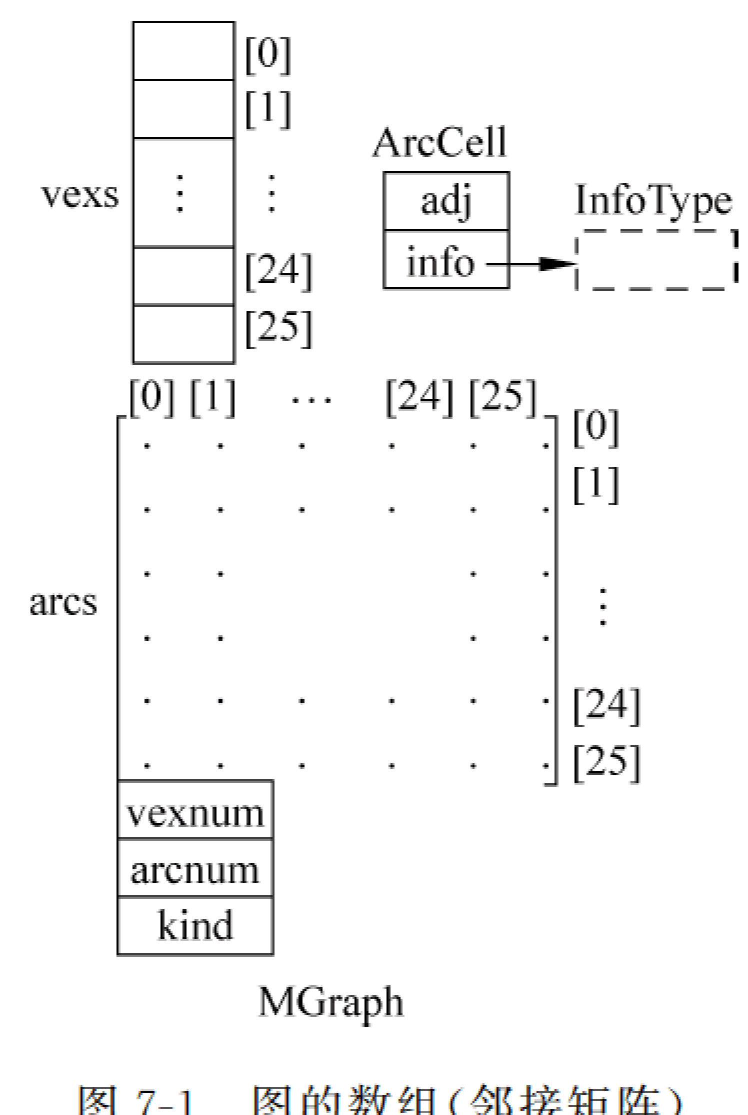


图 7-1 图的数组(邻接矩阵)存储结构

异,最简单的 VertexType 类型只包括一个成员:顶点名称。func7-1.cpp 定义了最简单的顶点类型和对这种顶点类型的输入输出操作。

```
// func7-1.cpp 包括顶点信息类型的定义及对它的操作
#define MAX_NAME 9 // 顶点名称字符串的最大长度+1
struct VertexType // 最简单的顶点信息类型(只有顶点名称)
{ char name[MAX_NAME]; // 顶点名称
};
void Visit(VertexType ver) // 与之配套的访问顶点的函数
{ printf("%s",ver.name);
}
void Input(VertexType &ver) // 与之配套的输入顶点信息的函数
{ scanf("%s",ver.name);
}
void InputFromFile(FILE *f,VertexType &ver)
// 与之配套的从文件输入顶点信息的函数
{ fscanf(f,"%s",ver.name);
}
```

MGraph 中存储弧(边)信息的类型是 AdjMatrix 类型,它是一个二维数组,用来存储任意 2 顶点之间的弧(边)信息。数组的元素类型是 ArcCell 类型,它是结构体类型,包括 VRType 和 InfoType * 类型。VRType 是数值型的,一般是整型,存储 0、1、∞或权值;InfoType 一般是结构体类型,存储弧(边)的相关信息,如果是文字信息,InfoType * 就应是 char * 类型。结构简单的弧(边),也可能没有相关信息,则要设指针为空。func7-2.cpp 定义了弧(边)的相关信息类型为字符型和对这种类型的输入输出操作。

```
// func7-2.cpp 包括弧(边)的相关信息类型的定义及对它的操作
#define MAX_INFO 20 // 弧(边)的相关信息字符串的最大长度+1
typedef char InfoType; // 弧(边)的相关信息类型
void InputArc(InfoType *&arc) // 与之配套的输入弧(边)的相关信息的函数
{ char s[MAX_INFO]; // 临时存储空间
  int m;
  printf("请输入该弧(边)的相关信息(<%d个字符):",MAX_INFO);
  gets(s); // 输入字符串(可包括空格)
  m=strlen(s); // 字符串长度
  if(m) // 长度不为0
  { arc=(char*)malloc((m+1)*sizeof(char)); // 动态生成相关信息存储空间
    strcpy(arc,s); // 复制 s 到 arc
  }
}
void InputArcFromFile(FILE*f,InfoType*&arc) // 由文件输入弧(边)的相关信息的函数
{ char s[MAX_INFO]; // 临时存储空间
  fgets(s,MAX_INFO,f); // 由文件输入字符串(可包括空格)
  arc=(char*)malloc((strlen(s)+1)*sizeof(char)); // 动态生成相关信息存储空间
```

```
  strcpy(arc,s); // 复制 s 到 arc
}
void OutputArc(InfoType* arc) // 与之配套的输出弧(边)的相关信息的函数
{ printf("%s\n",arc);
}
```

图 7-2 是根据 c7-1.h 定义的有向图的存储示例。vexs[]数组存放各顶点的信息,arcs[][]数组存放各顶点邻接关系(是否互为邻接点)信息,如果 1 条弧从第 i 个顶点发出,终止于第 j 个顶点,则 arcs[i][j]=1。以图 7-2(b)为例,arcs[0][1]=1,说明从 v1 到 v2 有 1 条弧。设对角元素(arcs[i][i])的邻接关系为 0,则 arcs[][]数组中值为 1 的元素的个数等于有向图的弧数。图 7-3 是根据 c7-1.h 定义的无向网(网也称为带权图)的存储示例。同图 7-2 一样,图 7-3 中的 vexs[]数组仍存放各顶点的信息,arcs[][]数组存放各顶点邻接关系信息。对于网,顶点互为邻接点,则其值为权值;否则其值为∞。设对角元素(arcs[i][i])的邻接关系为∞。如果在第 i 个顶点和第 j 个顶点之间有边(无向),则 arcs[i][j]= arcs[j][i]=权值。以图 7-3(b)为例,arcs[0][1]= arcs[1][0]=3,说明在 v1、v2 之间有 1 条边,其权值为 3。无向图或网的二维数组是以主对角线为轴对称的,对称的两个单元表示同一条边。arcs[][]数组中值不为∞的元素的个数等于无向网边数的 2 倍。

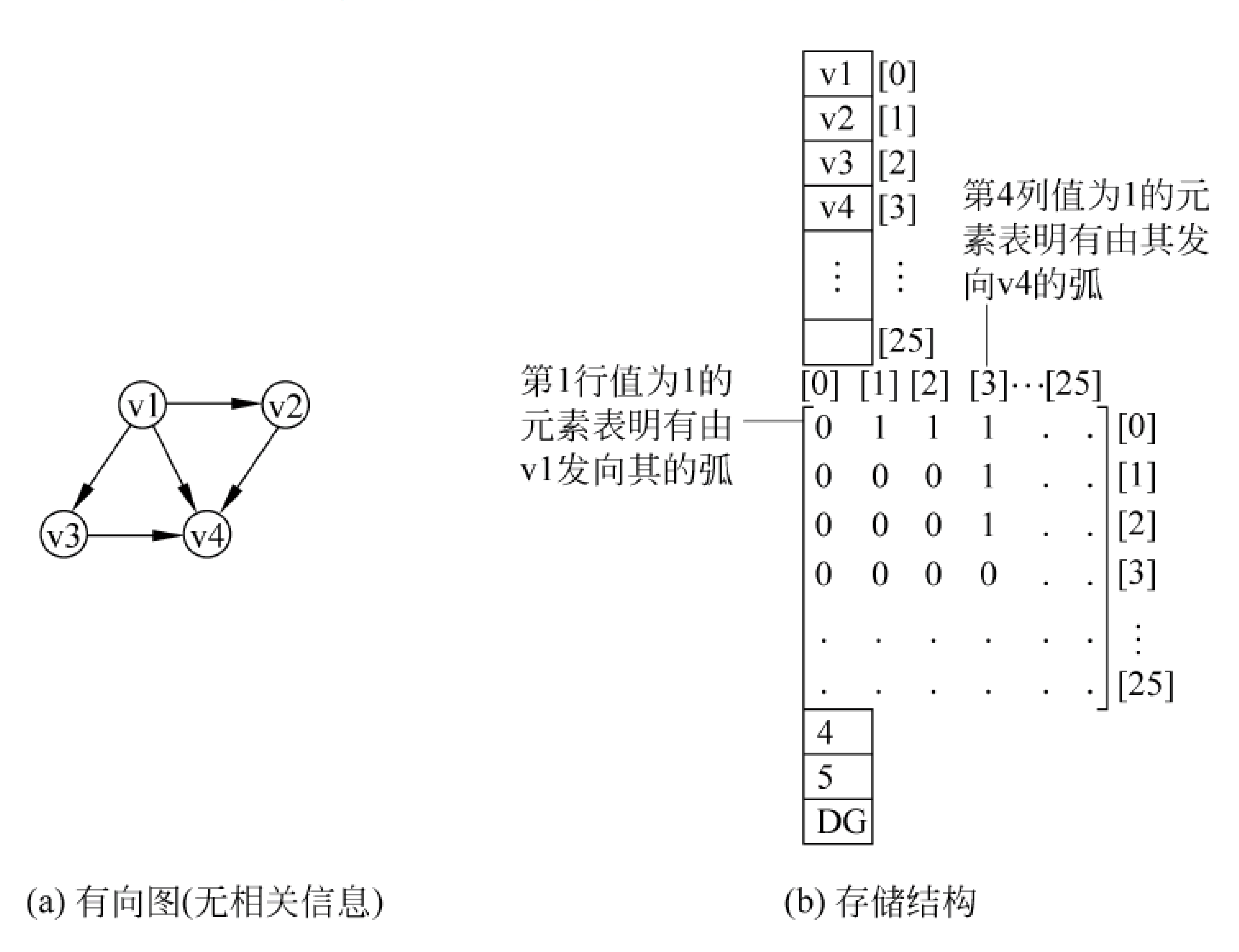
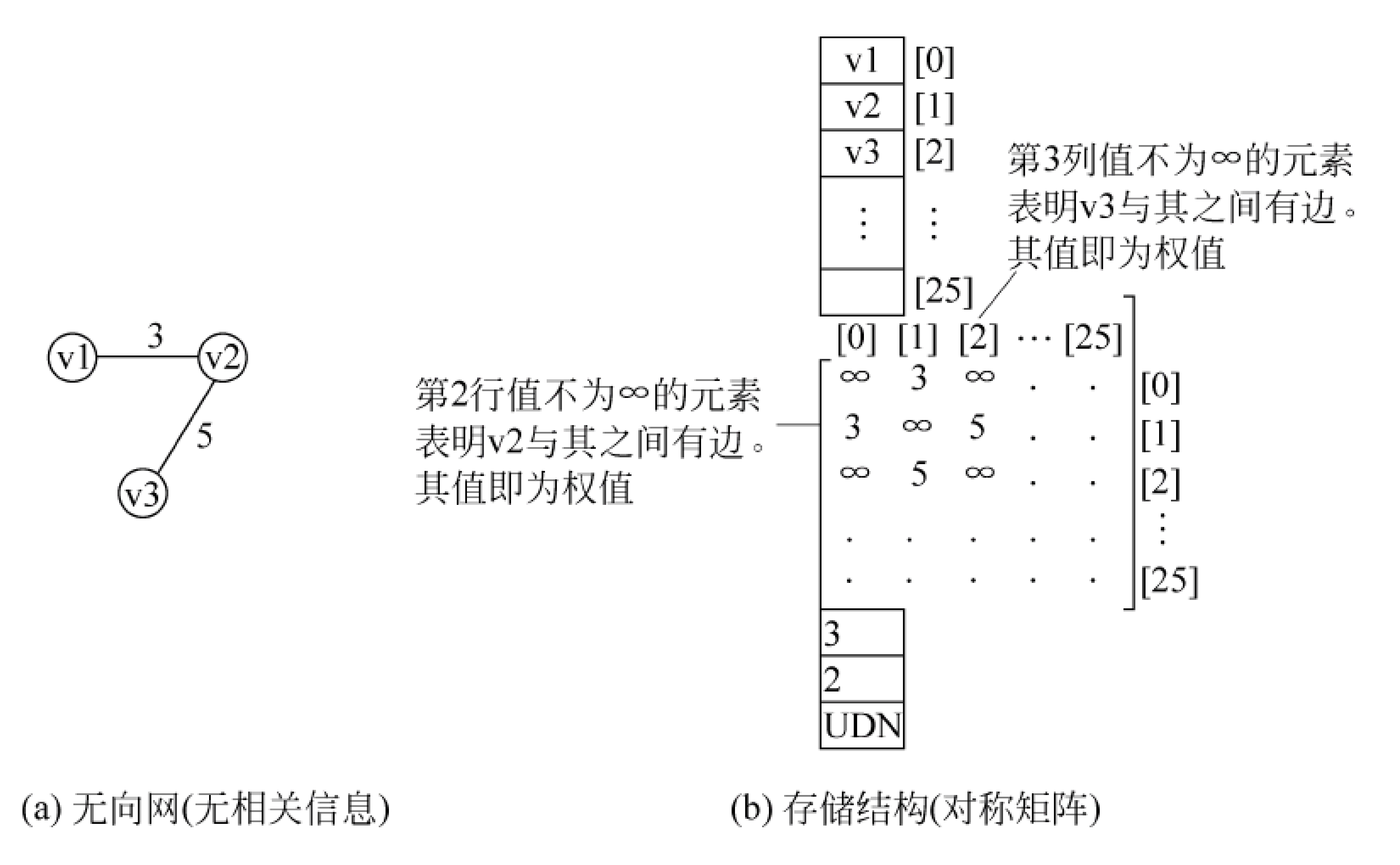



图 7-2 有向图的数组(邻接矩阵)存储示例

在这种数组(邻接矩阵)存储结构中,数组所占用的存储空间与图的弧(边)数无关,从节约存储空间方面考虑,它适用于边数较多的稠密图。

```
// bo7-1.cpp 图的邻接矩阵存储(存储结构由 c7-1.h 定义)的基本操作(17 个),包括算法 7.1 和算法 7.2
int LocateVex(MGraph G,VertexType u)
{ // 初始条件:图 G 存在,u 和 G 中顶点有相同特征(顶点名称相同)
  // 操作结果:若 G 中存在顶点 u,则返回该顶点在图中位置(序号);否则返回 -1
  int i;
  for(i = 0;i<G.vexnum;++i) // 对于所有顶点依次查找
    if(strcmp(u.name,G.vexs[i].name) == 0) // 顶点与给定的 u 的顶点名称相同
```




(a) 无向网(无相关信息)　　(b) 存储结构(对称矩阵)

图 7-3　无向网的数组(邻接矩阵)存储示例

```
      return i; // 返回顶点序号
  return -1; // 图 G 中不存在与顶点 u 有相同名称的顶点
}
void CreateDG(MGraph &G)
{ // 采用数组(邻接矩阵)表示法,构造有向图 G
  int i,j,k,IncInfo; // IncInfo 为 0 则弧不含相关信息
  VertexType v1,v2; // 顶点类型
  printf("请输入有向图 G 的顶点数,弧数,弧是否含相关信息(是:1 否:0):");
  scanf("%d,%d,%d",&G.vexnum,&G.arcnum,&IncInfo);
  printf("请输入%d个顶点的值(名称<%d个字符):\n",G.vexnum,MAX_NAME);
  for(i=0;i<G.vexnum;++i) // 构造顶点向量
    Input(G.vexs[i]); // 根据顶点信息的类型,输入顶点信息,在 func7-1.cpp 中
  for(i=0;i<G.vexnum;++i) // 初始化二维邻接矩阵(弧(边)信息)
    for(j=0;j<G.vexnum;++j)
    { G.arcs[i][j].adj=0; // 图,不相邻
      G.arcs[i][j].info=NULL; // 无相关信息
    }
  printf("请输入%d条弧的弧尾 弧头:\n",G.arcnum);
  for(k=0;k<G.arcnum;++k)
  { scanf("%s%s%*c",v1.name,v2.name); // %*c 吃掉回车符
    i=LocateVex(G,v1); // 弧尾的序号
    j=LocateVex(G,v2); // 弧头的序号
    G.arcs[i][j].adj=1; // 有向图
    if(IncInfo) // 有相关信息
      InputArc(G.arcs[i][j].info);
      // 动态生成存储空间,输入弧的相关信息,在 func7-2.cpp 中
  }
  G.kind=DG;
}
```

```
void CreateDN(MGraph &G)
{ // 采用数组(邻接矩阵)表示法,构造有向网 G
  int i,j,k,IncInfo; // IncInfo 为 0 则弧不含相关信息
  VRType w; // 顶点关系类型
  VertexType v1,v2; // 顶点类型
  printf("请输入有向网 G 的顶点数,弧数,弧是否含相关信息(是:1 否:0):");
  scanf("%d,%d,%d",&G.vexnum,&G.arcnum,&IncInfo);
  printf("请输入%d个顶点的值(名称<%d个字符):\n",G.vexnum,MAX_NAME);
  for(i=0;i<G.vexnum;++i) // 构造顶点向量
    Input(G.vexs[i]); // 根据顶点信息的类型,输入顶点信息,在 func7-1.cpp 中
  for(i=0;i<G.vexnum;++i) // 初始化二维邻接矩阵
    for(j=0;j<G.vexnum;++j)
    { G.arcs[i][j].adj=INFINITY; // 网,不相邻
      G.arcs[i][j].info=NULL; // 无相关信息
    }
  printf("请输入%d条弧的弧尾 弧头 权值:\n",G.arcnum);
  for(k=0;k<G.arcnum;++k)
  { scanf("%s%s%d%*c",v1.name,v2.name,&w); // %*c 吃掉回车符
    i=LocateVex(G,v1); // 弧尾的序号
    j=LocateVex(G,v2); // 弧头的序号
    G.arcs[i][j].adj=w; // 有向网
    if(IncInfo) // 有相关信息
      InputArc(G.arcs[i][j].info);
      // 动态生成存储空间,输入弧的相关信息,在 func7-2.cpp 中
  }
  G.kind=DN;
}
void CreateUDG(MGraph &G)
{ // 采用数组(邻接矩阵)表示法,构造无向图 G
  int i,j,k,IncInfo; // IncInfo 为 0 则弧不含相关信息
  VertexType v1,v2; // 顶点类型
  printf("请输入无向图 G 的顶点数,边数,边是否含相关信息(是:1 否:0):");
  scanf("%d,%d,%d",&G.vexnum,&G.arcnum,&IncInfo);
  printf("请输入%d个顶点的值(名称<%d个字符):\n",G.vexnum,MAX_NAME);
  for(i=0;i<G.vexnum;++i) // 构造顶点向量
    Input(G.vexs[i]); // 根据顶点信息的类型,输入顶点信息,在 func7-1.cpp 中
  for(i=0;i<G.vexnum;++i) // 初始化二维邻接矩阵(弧(边)信息)
    for(j=0;j<G.vexnum;++j)
    { G.arcs[i][j].adj=0; // 图,不相邻
      G.arcs[i][j].info=NULL; // 无相关信息
    }
  printf("请输入%d条边的顶点1 顶点2:\n",G.arcnum);
  for(k=0;k<G.arcnum;++k)
  { scanf("%s%s%*c",v1.name,v2.name); // %*c 吃掉回车符
```

```
    i = LocateVex(G,v1); // 顶点 1 的序号
    j = LocateVex(G,v2); // 顶点 2 的序号
    G.arcs[i][j].adj  = 1; // 图
    if(IncInfo) // 有相关信息
      InputArc(G.arcs[i][j].info);
      // 动态生成存储空间,输入弧的相关信息,在 func7-2.cpp 中
    G.arcs[j][i]  = G.arcs[i][j]; // 无向,两个单元的信息相同
  }
  G.kind = UDG;
}
void CreateUDN(MGraph &G)
{ // 采用数组(邻接矩阵)表示法,构造无向网 G。算法 7.2
  int i,j,k,IncInfo; // IncInfo 为 0 则弧不含相关信息
  VRType w; // 顶点关系类型
  VertexType v1,v2; // 顶点类型
  printf("请输入无向网 G 的顶点数,边数,边是否含相关信息(是:1 否:0):");
  scanf("%d,%d,%d",&G.vexnum,&G.arcnum,&IncInfo);
  printf("请输入%d 个顶点的值(名称<%d 个字符):\n",G.vexnum,MAX_NAME);
  for(i = 0;i<G.vexnum;++i) // 构造顶点向量
    Input(G.vexs[i]); // 根据顶点信息的类型,输入顶点信息,在 func7-1.cpp 中
  for(i = 0;i<G.vexnum;++i) // 初始化二维邻接矩阵
    for(j = 0;j<G.vexnum;++j)
    { G.arcs[i][j].adj = INFINITY; // 网,不相邻
      G.arcs[i][j].info = NULL; // 无相关信息
    }
  printf("请输入%d 条边的顶点 1 顶点 2 权值:\n",G.arcnum);
  for(k = 0;k<G.arcnum;++k)
  { scanf("%s%s%d%*c",v1.name,v2.name,&w); // %*c 吃掉回车符
    i = LocateVex(G,v1); // 顶点 1 的序号
    j = LocateVex(G,v2); // 顶点 2 的序号
    G.arcs[i][j].adj = w; // 网
    if(IncInfo) // 有相关信息
      InputArc(G.arcs[i][j].info);
      // 动态生成存储空间,输入弧的相关信息,在 func7-2.cpp 中
    G.arcs[j][i] = G.arcs[i][j]; // 无向,两个单元的信息相同
  }
  G.kind = UDN;
}
void CreateGraph(MGraph &G)
{ // 采用数组(邻接矩阵)表示法,构造图 G。修改算法 7.1
  printf("请输入图 G 的类型(有向图:0 有向网:1 无向图:2 无向网:3):");
  scanf("%d",&G.kind);
  switch(G.kind) // 根据图 G 的类型,调用不同的构造图的函数
  { case  DG:CreateDG(G); // 构造有向图
```

```
            break;
        case  DN:CreateDN(G); // 构造有向网
            break;
        case UDG:CreateUDG(G); // 构造无向图
            break;
        case UDN:CreateUDN(G); // 构造无向网
    }
}
VertexType GetVex(MGraph G,int v)
{ // 初始条件:图 G 存在,v 是 G 中某个顶点的序号。操作结果:返回 v 的值
  if(v>= G.vexnum||v<0) // 图 G 中不存在序号为 v 的顶点
    exit(OVERFLOW);
  return G.vexs[v]; // 返回该顶点的信息
}
Status PutVex(MGraph &G,VertexType v,VertexType value)
{ // 初始条件:图 G 存在,v 是 G 中某个顶点。操作结果:对 v 赋新值 value
  int k = LocateVex(G,v); // k 为顶点 v 在图 G 中的序号
  if(k<0) // 不存在顶点 v
    return ERROR;
  G.vexs[k] = value; // 将新值赋给顶点 v(其序号为 k)
  return OK;
}
int FirstAdjVex(MGraph G,int v)
{ // 初始条件:图 G 存在,v 是 G 中某个顶点的序号
  // 操作结果:返回 v 的第 1 个邻接顶点的序号。若顶点在 G 中没有邻接顶点,则返回 -1
  int i;
  VRType j = 0; // 顶点关系类型,图
  if(G.kind%2) // 网
    j = INFINITY;
  for(i = 0;i<G.vexnum;i++) // 从第 1 个顶点开始查找
    if(G.arcs[v][i].adj!= j) // 是第 1 个邻接顶点
      return i; // 返回该邻接顶点的序号
  return -1; // 没有邻接顶点
}
int NextAdjVex(MGraph G,int v,int w)
{ // 初始条件:图 G 存在,v 是 G 中某个顶点的序号,w 是 v 的邻接顶点的序号
  // 操作结果:返回 v 的(相对于 w 的)下一个邻接顶点的序号,
  //           若 w 是 v 的最后一个邻接顶点,则返回 -1
  int i;
  VRType j = 0; // 顶点关系类型,图
  if(G.kind%2) // 网
    j = INFINITY;
  for(i = w+1;i<G.vexnum;i++) // 从第 w+1 个顶点开始查找
    if(G.arcs[v][i].adj!= j) // 是从 w+1 开始的第 1 个邻接顶点
```

```
        return i; // 返回该邻接顶点的序号
    return -1; // 没有下一个邻接顶点
}
void InsertVex(MGraph &G,VertexType v)
{ // 初始条件：图 G 存在,v 和图 G 中顶点有相同特征
  // 操作结果：在图 G 中增添新顶点 v(不增添与顶点相关的弧,留待 InsertArc()去做)
    int i;
    VRType j = 0; // 顶点关系类型,图
    if(G.kind%2) // 网
      j = INFINITY;
    G.vexs[G.vexnum] = v; // 将值 v 赋给新顶点
    for(i = 0;i<= G.vexnum;i++) // 对于新增行、新增列
    { G.arcs[G.vexnum][i].adj = G.arcs[i][G.vexnum].adj = j;
       // 初始化新增行、新增列邻接矩阵的值(无边或弧)
      G.arcs[G.vexnum][i].info = G.arcs[i][G.vexnum].info = NULL; // 初始化相关信息指针
    }
    G.vexnum++; // 图 G 的顶点数加 1
}
Status InsertArc(MGraph &G,VertexType v,VertexType w)
{ // 初始条件：图 G 存在,v 和 w 是 G 中两个顶点
  // 操作结果：在 G 中增添弧<v,w>,若 G 是无向的,则还增添对称弧<w,v>
    int i,v1,w1;
    v1 = LocateVex(G,v); // 弧尾顶点 v 的序号
    w1 = LocateVex(G,w); // 弧头顶点 w 的序号
    if(v1<0||w1<0) // 不存在顶点 v 或 w
      return ERROR;
    G.arcnum++; // 弧或边数加 1
    if(G.kind%2) // 网
    { printf("请输入此弧或边的权值：");
      scanf("%d",&G.arcs[v1][w1].adj);
    }
    else // 图
      G.arcs[v1][w1].adj = 1;
    printf("是否有该弧或边的相关信息(0：无 1：有)：");
    scanf("%d%*c",&i);
    if(i)
      InputArc(G.arcs[v1][w1].info); // 动态生成存储空间,输入弧的相关信息,在 func7-2.cpp 中
    if(G.kind>1) // 无向
      G.arcs[w1][v1] = G.arcs[v1][w1]; // 有同样的邻接值,指向同一个相关信息
    return OK;
}
Status DeleteArc(MGraph &G,VertexType v,VertexType w)
{ // 初始条件：图 G 存在,v 和 w 是 G 中两个顶点
  // 操作结果：在 G 中删除弧<v,w>,若 G 是无向的,则还删除对称弧<w,v>
```

```
    int v1,w1;
    VRType j = 0; // 顶点关系类型,图
    if(G.kind%2) // 网
      j = INFINITY;
    v1 = LocateVex(G,v); // 弧尾顶点 v 的序号
    w1 = LocateVex(G,w); // 弧头顶点 w 的序号
    if(v1<0||w1<0) // 不存在顶点 v 或 w
      return ERROR;
    if(G.arcs[v1][w1].adj!= j) // 有弧<v,w>
    { G.arcs[v1][w1].adj = j; // 删除弧<v,w>
      if(G.arcs[v1][w1].info) // 有相关信息
      { free(G.arcs[v1][w1].info); // 释放相关信息
        G.arcs[v1][w1].info = NULL; // 置相关信息指针为空
      }
      if(G.kind>= 2) // 无向,删除对称弧<w,v>
        G.arcs[w1][v1] = G.arcs[v1][w1]; // 删除弧,置相关信息指针为空
      G.arcnum--; // 弧数 - 1
    }
    return OK;
}
Status DeleteVex(MGraph &G,VertexType v)
{ // 初始条件:图 G 存在,v 是 G 中某个顶点。操作结果:删除 G 中顶点 v 及其相关的弧
    int i,j,k;
    k = LocateVex(G,v); // k 为待删除顶点 v 的序号
    if(k<0) // v 不是图 G 的顶点
      return ERROR;
    for(i = 0;i<G.vexnum;i++)
      DeleteArc(G,v,G.vexs[i]); // 删除由顶点 v 发出的所有弧
    if (G.kind<2) // 有向
      for(i = 0;i<G.vexnum;i++)
        DeleteArc(G,G.vexs[i],v); // 删除发向顶点 v 的所有弧
      for(j = k + 1;j<G.vexnum;j++)
        G.vexs[j - 1] = G.vexs[j]; // 序号 k 后面的顶点向量依次前移
      for(i = 0;i<G.vexnum;i++)
        for(j = k + 1;j<G.vexnum;j++)
          G.arcs[i][j - 1] = G.arcs[i][j]; // 移动待删除顶点之右的矩阵元素
      for(i = 0;i<G.vexnum;i++)
        for(j = k + 1;j<G.vexnum;j++)
          G.arcs[j - 1][i] = G.arcs[j][i]; // 移动待删除顶点之下的矩阵元素
      G.vexnum--; // 更新图的顶点数
      return OK;
}
void DestroyGraph(MGraph &G)
{ // 初始条件:图 G 存在。操作结果:销毁图 G
    int i;
```

```
  for(i = G.vexnum - 1;i>= 0;i-- ) // 由大到小逐一删除顶点及与其相关的弧(边)
    DeleteVex(G,G.vexs[i]);
}
void Display(MGraph G)
{ // 输出邻接矩阵存储结构的图 G
  int i,j;
  char s[7] = "无向网",s1[3] = "边";
  switch(G.kind)
  { case  DG:strcpy(s,"有向图");
             strcpy(s1,"弧");
             break;
    case  DN:strcpy(s,"有向网");
             strcpy(s1,"弧");
             break;
    case UDG:strcpy(s,"无向图");
    case UDN:;
  }
  printf("%d 个顶点 %d 条 %s 的 %s。顶点依次是：",G.vexnum,G.arcnum,s1,s);
  for(i = 0;i<G.vexnum;++i)
    Visit(GetVex(G,i)); // 根据顶点信息的类型,访问第 i 个顶点,在 func7-1.cpp 中
  printf("\nG.arcs.adj: \n");
  for(i = 0;i<G.vexnum;i++ ) // 输出二维数组 G.arcs.adj
  { for(j = 0;j<G.vexnum;j++ )
      printf("%11d",G.arcs[i][j].adj);
    printf("\n");
  }
  printf("G.arcs.info: \n"); // 输出 G.arcs.info
  if(G.kind<2) // 有向
    printf(" 弧尾  弧头 该 %s 的信息：\n",s1);
  else // 无向
    printf("顶点 1 顶点 2 该 %s 的信息：\n",s1);
  for(i = 0;i<G.vexnum;i++ )
    if(G.kind<2) // 有向
    { for(j = 0;j<G.vexnum;j++ )
        if(G.arcs[i][j].info)
        { printf("%5s %5s ",G.vexs[i].name,G.vexs[j].name);
          OutputArc(G.arcs[i][j].info); // 输出弧(边)的相关信息,在 func7-2.cpp 中
        }
    } // 加括号为避免 if-else 对配错
    else // 无向,输出上三角
      for(j = i + 1;j<G.vexnum;j++ )
        if(G.arcs[i][j].info)
        { printf("%5s%5s",G.vexs[i].name,G.vexs[j].name);
          OutputArc(G.arcs[i][j].info); // 输出弧(边)的相关信息,在 func7-2.cpp 中
        }
}
void CreateFromFile(MGraph &G,char* filename,int IncInfo)
```

```
{ // 采用数组(邻接矩阵)表示法,由文件构造图或网 G。IncInfo = 0 或 1,表示弧(边)无或有相关信息
  int i,j,k;
  VRType w = 0; // 顶点关系类型,图
  VertexType v1,v2; // 顶点类型
  FILE *f; // 文件指针类型
  f = fopen(filename,"r"); // 打开数据文件,并以 f 表示
  fscanf(f,"%d",&G.kind); // 由文件输入 G 的类型
  if(G.kind%2) // 网
    w = INFINITY;
  fscanf(f,"%d",&G.vexnum); // 由文件输入 G 的顶点数
  for(i = 0;i<G.vexnum;++i)
    InputFromFile(f,G.vexs[i]); // 由文件输入顶点信息,在 func7-1.cpp 中
  fscanf(f,"%d",&G.arcnum); // 由文件输入 G 的弧(边)数
  for(i = 0;i<G.vexnum;++i) // 初始化二维邻接矩阵
    for(j = 0;j<G.vexnum;++j)
    { G.arcs[i][j].adj = w; // 不相邻
      G.arcs[i][j].info = NULL; // 没有相关信息
    }
  if(!(G.kind%2)) // 图
    w = 1;
  for(k = 0;k<G.arcnum;++k) // 对于所有弧
  { fscanf(f,"%s%s",v1.name,v2.name); // 输入弧尾、弧头的名称
    if(G.kind%2) // 网
      fscanf(f,"%d",&w); // 再输入权值
    i = LocateVex(G,v1); // 弧尾的序号
    j = LocateVex(G,v2); // 弧头的序号
    G.arcs[i][j].adj = w; // 权值
    if(IncInfo) // 有相关信息
      InputArcFromFile(f,G.arcs[i][j].info);
      // 由文件动态生成存储空间,输入弧的相关信息,在 func7-2.cpp 中
    if(G.kind>1) // 无向
      G.arcs[j][i] = G.arcs[i][j]; // 无向,两个单元的信息相同
  }
  fclose(f); // 关闭数据文件
}

// func7-3.cpp 用于检验邻接矩阵和邻接表的主函数
void main()
{
  int i,j,k,n;
  char s[3] = "边";
  Graph g; // 抽象的图类型
  VertexType v1,v2; // 顶点类型
  printf("请依次选择有向图,有向网,无向图,无向网:\n");
  for(i = 0;i<4;i++) // 验证 4 种情况
  { CreateGraph(g); // 构造图 g
    Display(g); // 输出图 g
```

```
    printf("插入新顶点,请输入新顶点的值：");
    Input(v1); // 根据顶点信息的类型,输入顶点 v1 的信息,在 func7-1.cpp 中
    InsertVex(g,v1); // 在图 g 中插入顶点 v1
    if(g.kind<2) // 有向
      strcpy(s,"弧");
    printf("插入与新顶点有关的%s,请输入%s数：",s,s);
    scanf("%d",&n);
    for(k=0;k<n;k++) // 依次插入 n 条弧(边)
    { printf("请输入另一顶点的名称：");
      scanf("%s",v2.name);
      if(g.kind<=1) // 有向
      { printf("请输入另一顶点的方向(0：弧头 1：弧尾)：");
        scanf("%d",&j);
        if(j) // v2 是弧尾
          InsertArc(g,v2,v1); // 在图 g 中插入弧 v2→v1
        else // v2 是弧头
          InsertArc(g,v1,v2); // 在图 g 中插入弧 v1→v2
      }
      else // 无向
        InsertArc(g,v1,v2); // 在图 g 中插入边 v1—v2
    }
    Display(g); // 输出图 g
    printf("删除顶点及相关的%s,请输入待删除顶点的名称：",s);
    scanf("%s",v1.name);
    DeleteVex(g,v1); // 在图 g 中删除顶点 v1
    Display(g); // 输出图 g
    if(i==3) // 对于最后一个(无向网),测试以下函数
    { printf("修改顶点的值,请输入待修改顶点名称 新值：");
      scanf("%s",v1.name); // 输入待修改顶点名称,以查找待修改的顶点
      Input(v2); // 输入顶点的新值,以代替原值
      PutVex(g,v1,v2); // 将图 g 中顶点 v1 的值改为 v2
      if(g.kind<2) // 有向(假设最后一个可以不是无向网)
        printf("删除一条%s,请输入待删除%s的弧尾 弧头：",s,s);
      else // 无向
        printf("删除一条%s,请输入待删除%s的顶点 1 顶点 2：",s,s);
      scanf("%s%s",v1.name,v2.name); // 输入待删除弧(边)的 2 顶点的名称
      DeleteArc(g,v1,v2); // 删除图 g 中由顶点 v1 指向顶点 v2 的弧(边)
      Display(g); // 输出图 g
    }
    DestroyGraph(g); // 销毁图 g
  }
}
```

```
// main7-1.cpp 检验 bo7-1.cpp 的主程序
#include"c1.h"
#include"func7-1.cpp" // 包括顶点信息类型的定义及对它的操作
#include"func7-2.cpp" // 包括弧(边)的相关信息类型的定义及对它的操作
```

```
#include"c7-1.h" // 图的数组(邻接矩阵)存储结构
#include"bo7-1.cpp" // 图的数组(邻接矩阵)存储结构的基本操作(17个)
typedef MGraph Graph; // 定义 func7-3.cpp 中 Graph 的类型为 MGraph(邻接矩阵存储结构)
#include"func7-3.cpp" // 主函数
```

**程序运行结果：**

```
请依次选择有向图,有向网,无向图,无向网：
请输入图G的类型(有向图：0 有向网：1 无向图：2 无向网：3)：0↙
请输入有向图G的顶点数,弧数,弧是否含相关信息(是：1 否：0)：2,1,0↙
请输入2个顶点的值(名称<9个字符)：
a1 a2↙
请输入1条弧的弧尾 弧头：
a2 a1↙
2个顶点1条弧的有向图。顶点依次是：a1 a2（见图7-4）
G.arcs.adj：
          0          0
          1          0
G.arcs.info：
 弧尾   弧头 该弧的信息：
```

图7-4　不含相关信息的有向图

```
插入新顶点,请输入新顶点的值：a3↙
插入与新顶点有关的弧,请输入弧数：2↙
请输入另一顶点的名称：a1↙
请输入另一顶点的方向(0：弧头 1：弧尾)：0↙
是否有该弧或边的相关信息(0：无 1：有)：0↙
请输入另一顶点的名称：a2↙
请输入另一顶点的方向(0：弧头 1：弧尾)：1↙
是否有该弧或边的相关信息(0：无 1：有)：0↙
3个顶点3条弧的有向图。顶点依次是：a1 a2 a3（见图7-5）
G.arcs.adj：
          0          0          0
          1          0          1
          1          0          0
G.arcs.info：
 弧尾   弧头 该弧的信息：
```

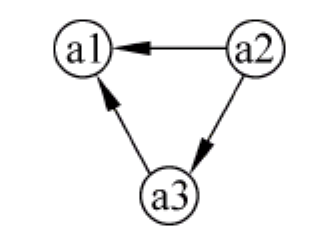



图7-5　插入顶点a3

```
删除顶点及相关的弧,请输入待删除顶点的名称：a1↙
2个顶点1条弧的有向图。顶点依次是：a2 a3（见图7-6）
G.arcs.adj：
          0          1
          0          0
G.arcs.info：
 弧尾   弧头 该弧的信息：
```

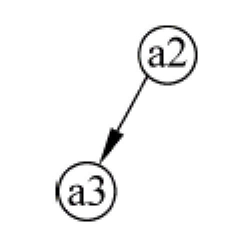



图7-6　删除顶点a1

```
请输入图G的类型(有向图：0 有向网：1 无向图：2 无向网：3)：1↙
请输入有向网G的顶点数,弧数,弧是否含相关信息(是：1 否：0)：2,1,1↙
请输入2个顶点的值(名称<9个字符)：
b1 b2↙
```

```
请输入1条弧的弧尾 弧头 权值:
b1 b2 3↙
请输入该弧(边)的相关信息(<20个字符):Good morning! ↙
2个顶点1条弧的有向网。顶点依次是:b1 b2 (见图7-7)
G.arcs.adj:
      32767          3
      32767      32767
G.arcs.info:
 弧尾   弧头 该弧的信息:
   b1    b2 Good morning!
插入新顶点,请输入新顶点的值:b3↙
插入与新顶点有关的弧,请输入弧数:2↙
请输入另一顶点的名称:b1↙
请输入另一顶点的方向(0:弧头 1:弧尾):0↙
请输入此弧或边的权值:5↙
是否有该弧或边的相关信息(0:无 1:有):1↙
请输入该弧(边)的相关信息(<20个字符):Good day! ↙
请输入另一顶点的名称:b2↙
请输入另一顶点的方向(0:弧头 1:弧尾):1↙
请输入此弧或边的权值:6↙
是否有该弧或边的相关信息(0:无 1:有):1↙
请输入该弧(边)的相关信息(<20个字符):Good bye! ↙
3个顶点3条弧的有向网。顶点依次是:b1 b2 b3 (见图7-8)
G.arcs.adj:
      32767          3      32767
      32767      32767          6
          5      32767      32767
G.arcs.info:
 弧尾   弧头 该弧的信息:
   b1    b2    Good morning!
   b2    b3    Good bye!
   b3    b1    Good day!
删除顶点及相关的弧,请输入待删除顶点的名称:b2↙
2个顶点1条弧的有向网。顶点依次是:b1 b3 (见图7-9)
G.arcs.adj:
      32767      32767
          5      32767
G.arcs.info:
 弧尾   弧头 该弧的信息:
   b3    b1    Good day!
请输入图G的类型(有向图:0 有向网:1 无向图:2 无向网:3):2↙
请输入无向图G的顶点数,边数,边是否含相关信息(是:1 否:0):2,1,1↙
请输入2个顶点的值(名称<9个字符):
c1 c2↙
```



图7-7 含相关信息的有向网



图7-8 插入顶点b3



图7-9 删除顶点b2

```
请输入 1 条边的顶点 1 顶点 2：
c1 c2↙
请输入该弧(边)的相关信息(<20 个字符)：good↙
2 个顶点 1 条边的无向图。顶点依次是：c1 c2（见图 7-10）
G.arcs.adj：
        0        1
        1        0
G.arcs.info：
顶点 1 顶点 2 该边的信息：
  c1   c2    good
插入新顶点，请输入新顶点的值：c3↙
插入与新顶点有关的弧，请输入弧数：2↙
请输入另一顶点的名称：c1↙
是否有该弧或边的相关信息(0：无 1：有)：1↙
请输入该弧(边)的相关信息(<20 个字符)：better↙
请输入另一顶点的名称：c2↙
是否有该弧或边的相关信息(0：无 1：有)：1↙
请输入该弧(边)的相关信息(<20 个字符)：best↙
3 个顶点 3 条边的无向图。顶点依次是：c1 c2 c3（见图 7-11）
G.arcs.adj：
        0        1        1
        1        0        1
        1        1        0
G.arcs.info：
顶点 1 顶点 2 该边的信息：
  c1   c2    good
  c1   c3    better
  c2   c3    best
删除顶点及相关的弧，请输入待删除顶点的名称：c3↙
2 个顶点 1 条边的无向图。顶点依次是：c1 c2（见图 7-12）
G.arcs.adj：
        0        1
        1        0
G.arcs.info：
顶点 1 顶点 2 该边的信息：
  c1     c2    good
请输入图 G 的类型(有向图：0 有向网：1 无向图：2 无向网：3)：3↙
请输入无向网 G 的顶点数，边数，边是否含相关信息(是：1 否：0)：2,1,0↙
请输入 2 个顶点的值(名称<9 个字符)：
d1 d2↙
请输入 1 条边的顶点 1 顶点 2 权值：
d1 d2 5↙
2 个顶点 1 条边的无向网。顶点依次是：d1 d2（见图 7-13）
G.arcs.adj：
```

c1 good c2

图 7-10 含相关信息的无向图

c1 good c2 better best c3

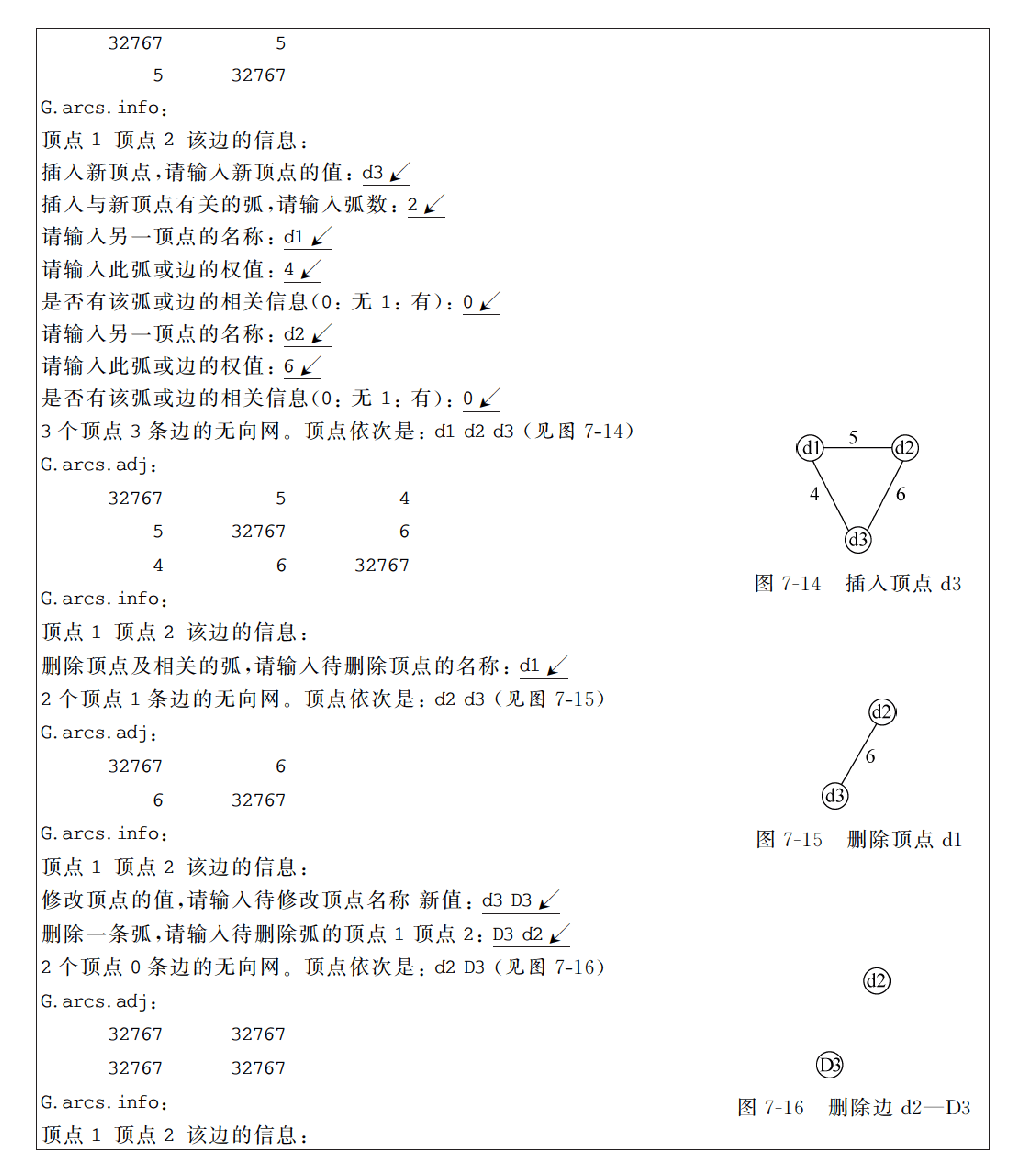
图 7-11 插入顶点 c3

c1 good c2

图 7-12 删除顶点 c3

d1 5 d2

图 7-13 不含相关信息的无向网



```
        32767           5
            5       32767
G.arcs.info:
顶点 1  顶点 2  该边的信息:
插入新顶点,请输入新顶点的值:d3↙
插入与新顶点有关的弧,请输入弧数:2↙
请输入另一顶点的名称:d1↙
请输入此弧或边的权值:4↙
是否有该弧或边的相关信息(0:无 1:有):0↙
请输入另一顶点的名称:d2↙
请输入此弧或边的权值:6↙
是否有该弧或边的相关信息(0:无 1:有):0↙
3 个顶点 3 条边的无向网。顶点依次是:d1 d2 d3 (见图 7-14)
G.arcs.adj:
        32767           5           4
            5       32767           6
            4           6       32767
G.arcs.info:
顶点 1  顶点 2  该边的信息:
删除顶点及相关的弧,请输入待删除顶点的名称:d1↙
2 个顶点 1 条边的无向网。顶点依次是:d2 d3 (见图 7-15)
G.arcs.adj:
        32767           6
            6       32767
G.arcs.info:
顶点 1  顶点 2  该边的信息:
修改顶点的值,请输入待修改顶点名称 新值:d3 D3↙
删除一条弧,请输入待删除弧的顶点 1 顶点 2:D3 d2↙
2 个顶点 0 条边的无向网。顶点依次是:d2 D3 (见图 7-16)
G.arcs.adj:
        32767       32767
        32767       32767
G.arcs.info:
顶点 1  顶点 2  该边的信息:
```



图 7-14　插入顶点 d3



图 7-15　删除顶点 d1



图 7-16　删除边 d2—D3

对于不同的问题,图的顶点和弧(边)的信息类型是不一样的。为了使 bo7-1.cpp 中的基本操作能够适用于各种图的顶点和弧(边)的信息类型,把图的顶点和弧(边)的信息类型从基本操作中抽出来,分别用 func7-1.cpp 和 func7-2.cpp 来定义图的顶点和弧(边)的信息类型及对它们的输入输出操作。当图的顶点和弧(边)的信息类型变化时,只须修改 func7-1.cpp 和 func7-2.cpp 即可。

## 7.1.2　邻接表

```
// c7-2.h 图的邻接表存储结构。在教科书第 163 页(见图 7-17)
#define MAX_VERTEX_NUM 20 // 最大顶点数
```

```
enum GraphKind{DG,DN,UDG,UDN}; // {有向图,有向网,无向图,无向网}
struct ArcNode // 表结点,存弧的信息
{ int adjvex; // 该弧所指向的顶点的位置(序号)
  InfoType * info; // 该弧相关信息(包括网的权值)的指针
  ArcNode * nextarc; // 指向下一条弧的指针
};
typedef struct // 头结点,存顶点的信息
{ VertexType data; // 顶点信息
  ArcNode * firstarc; // 第 1 个表结点的地址,指向第 1 条依附该顶点的弧的指针
}VNode,AdjList[MAX_VERTEX_NUM];
struct ALGraph // 邻接表结构
{ AdjList vertices; // 头结点(顶点)数组
  int vexnum,arcnum; // 图的当前顶点数和弧数
  GraphKind kind; // 图的种类标志
};
```

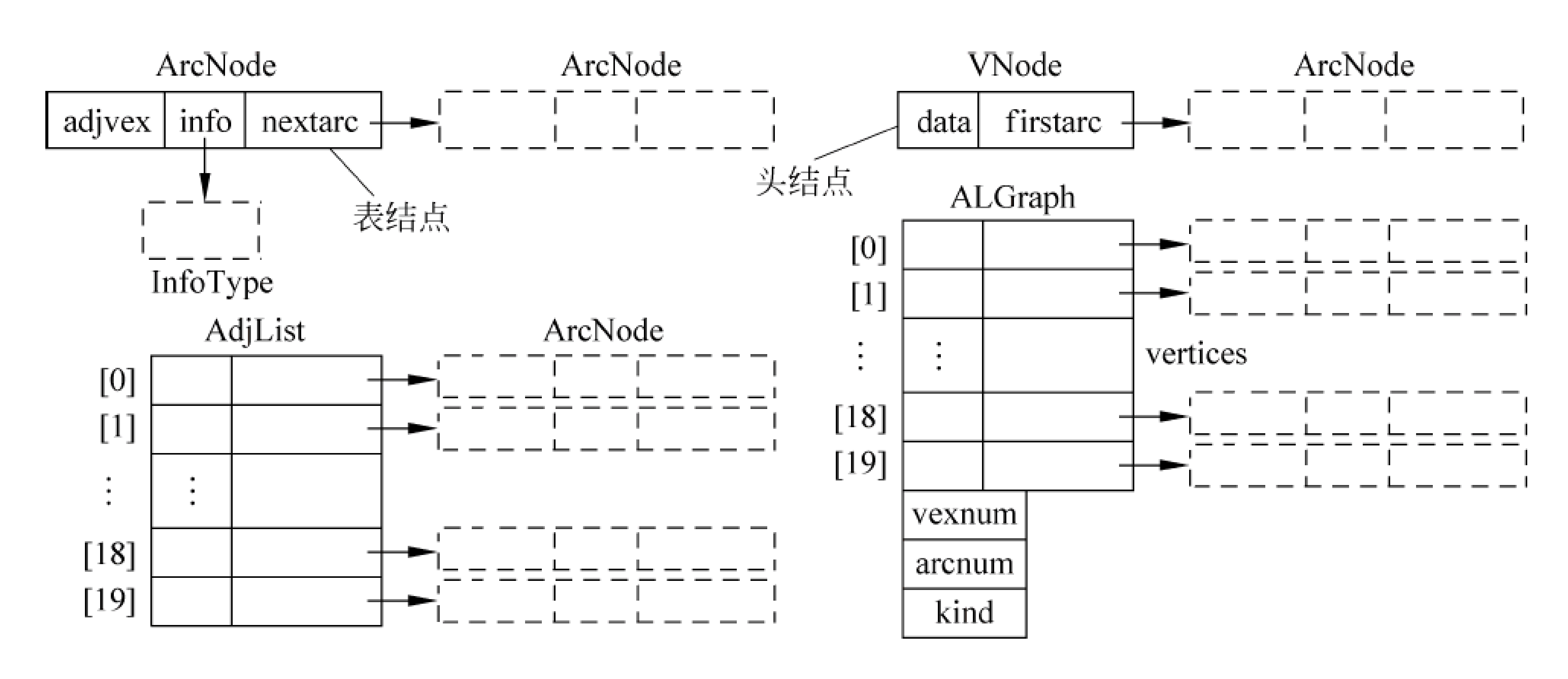



图 7-17 图的邻接表存储结构

和图的邻接矩阵存储结构一样,图的邻接表存储结构也用 VertexType 类型存储有关顶点的一切信息。如果图的 VertexType 类型只包括一个成员,即顶点名称,就可以仍然使用 func7-1. cpp 定义的顶点类型和对这种顶点类型的输入输出操作。

图的邻接表存储结构中的 InfoType 类型存储有关弧(边)的一切信息。它和图的邻接矩阵存储结构中的 InfoType 类型是有区别的。在 MGraph 中,弧(边)信息被分成 2 部分:一部分是顶点关系类型 VRType,其中根据不同情况分别存储 0、1、∞和权值;另一部分是 InfoType 类型,存储弧(边)除此之外的一切信息。而在图的邻接表存储结构中,0、1、∞根本不需要存储,权值和弧(边)的其他信息一起存储在 InfoType 类型中。结构简单的图,可以没有相关信息,则设指向 InfoType 类型的指针为空。网(带权图)的 InfoType 类型应是结构体,其中至少有权值项。func7-4. cpp 定义了仅有权值项的弧(边)的相关信息结构体类型和对这种类型的输入输出操作。

```
// func7-4.cpp 包括弧(边)的相关信息类型的定义及对它的操作
typedef int VRType; // 定义权值类型为整型
struct InfoType // 最简单的弧(边)的相关信息类型(只有权值)
```

```
{ VRType weight; // 权值
};
void InputArc(InfoType*&arc) // 与之配套的输入弧(边)的相关信息的函数
{ arc = (InfoType * )malloc(sizeof(InfoType)); // 动态生成存放弧(边)信息的空间
  scanf("%d",&arc->weight);
}
void OutputArc(InfoType* arc) // 与之配套的输出弧(边)的相关信息的函数
{ printf(":%d",arc->weight);
}
void InputArcFromFile(FILE* f,InfoType*&arc) // 由文件输入弧(边)的相关信息的函数
{ arc = (InfoType * )malloc(sizeof(InfoType)); // 动态生成存放弧(边)信息的空间
  fscanf(f,"%d",&arc->weight);
}
```

图 7-18 和图 7-19 分别是有向图和无向网的邻接表存储结构的示例。要注意的是，为了提高效率，bo7-2.cpp 中的基本操作函数 CreateGraph()产生链表时总是在表头插入结点。所以，对于给定的图，即使它的顶点输入顺序相同，邻接表的存储结构也不唯一。邻接表的存储结构还与弧或边的输入顺序有关。

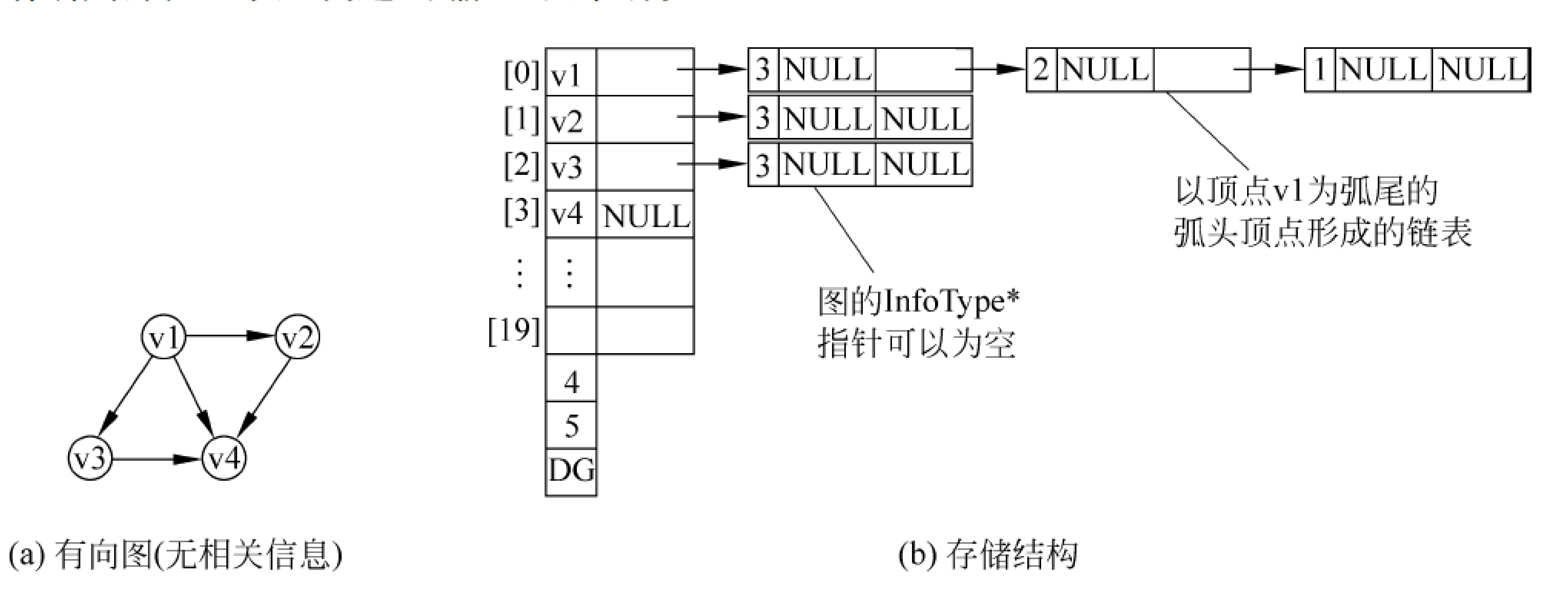



图 7-18　有向图的邻接表存储示例

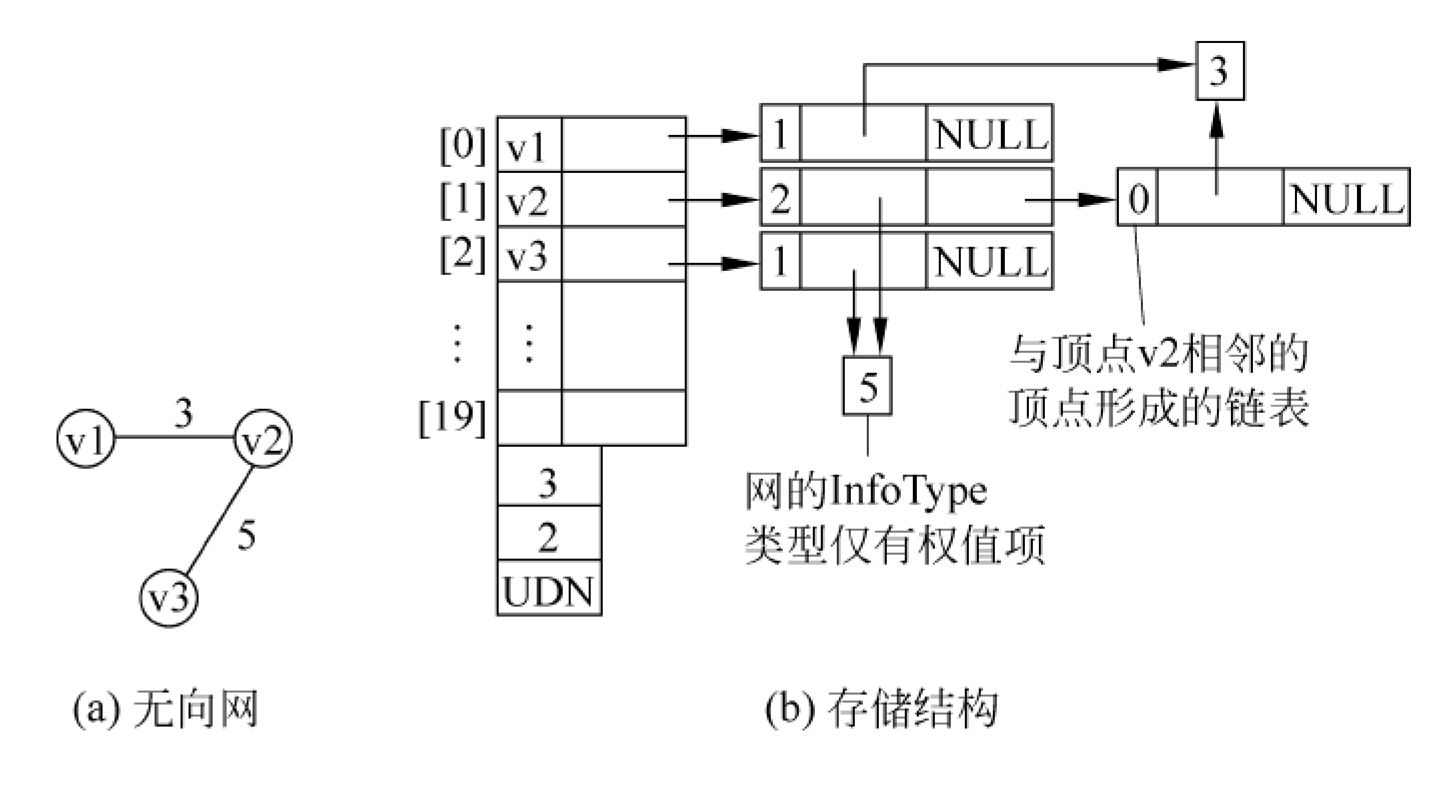



图 7-19　无向网的邻接表存储示例

对于无向的图或网，每条边产生 2 个表结点，分别在该边的 2 个顶点的链表上。由图 7-19 可见，2 条边的无向网有 4 个表结点。为简化，无向网的每条边只动态生成 1 个存放边的信

息(目前只是权值)的存储空间,由两个结点的指针共同指向。由于邻接表存储结构中的链表的长度与该顶点的邻接出弧或边数相等,显然,从节约存储空间方面考虑,图的邻接表存储结构适合存储弧或边相对较少的稀疏图。

由邻接表的存储结构可见,头结点数组的每个项由顶点信息和不带头结点的单链表组合而成。这样,对于单链表的处理,我们就可以利用不带头结点的单链表的基本操作(在bo2-3.cpp 中)和扩展操作(在 func2-4.cpp 中)来简化编程。为了能够利用这些操作,需要把在 bo2-3.cpp 和 func2-4.cpp 中用到的类型 ElemType、LNode、LinkList 和邻接表的类型 ArcNode 联系起来。c7-2'.h 就是根据 c7-2.h 建立了这种联系。

```
// c7-2'.h 图的邻接表存储结构(与单链表的变量类型建立联系)
#define MAX_VERTEX_NUM 20 // 最大顶点数
enum GraphKind{DG,DN,UDG,UDN}; // {有向图,有向网,无向图,无向网}
struct ElemType // 新增(见图 7-20)
{ int adjvex; // 该弧所指向的顶点的位置
  InfoType *info; // 该弧相关信息(包括网的权值)的指针
};
struct ArcNode // 表结点,存弧的信息,修改(见图 7-21)
{ ElemType data; // 除指针以外的部分都属于 ElemType
  ArcNode *nextarc; // 指向下一条弧的指针
};
typedef struct // 头结点,存顶点的信息
{ VertexType data; // 顶点信息
  ArcNode *firstarc; // 第 1 个表结点的地址,指向第 1 条依附该顶点的弧的指针
}VNode,AdjList[MAX_VERTEX_NUM];
struct ALGraph // 邻接表结构
{ AdjList vertices; // 头结点(顶点)数组
  int vexnum,arcnum; // 图的当前顶点数和弧数
  GraphKind kind; // 图的种类标志
};
#define LNode ArcNode // 新增,定义单链表的结点类型是图的表结点的类型
#define next nextarc // 新增,定义单链表结点的指针域是表结点指向下一条弧的指针域
typedef ArcNode *LinkList; // 新增,定义指向单链表结点的指针是指向图的表结点的指针
```



图 7-20 ElemType 类型



图 7-21 ArcNode 类型

```
// func2-4.cpp 不设头结点的单链表(存储结构由 c2-2.h 定义)的扩展操作(1 个),bo7-2.cpp 用到
LinkList Point(LinkList L,ElemType e,Status(*equal)(ElemType,ElemType),LinkList &p)
{ // 查找表 L 中与 e 满足 equal()条件的结点。如找到,返回指向该结点的指针,p 指向该结点的前驱
  // (若该结点是首元结点,则 p=NULL)。如表 L 中无满足条件的结点,则返回 NULL,p 无定义
  int j,i=LocateElem(L,e,equal); // 查找表 L 中与 e 满足 equal()条件的结点
  if(i) // 找到
  { if(i==1) // 是首元结点
    { p=NULL;
      return L;
    }
    p=L; // 不是首元结点,则 p 指向第 1 个结点
    for(j=2;j<i;j++) // p 指向所找结点的前驱
```

```
      p = p->next;
    return p->next; // 返回所找结点的指针
  }
  return NULL; // 未找到
}
```

```
// bo7-2.cpp 图的邻接表存储(存储结构由 c7-21.h 定义)的基本操作(14 个)
#include"bo2-3.cpp" // 不带头结点的单链表基本操作
#include"func2-4.cpp" // 不带头结点的单链表扩展操作
int LocateVex(ALGraph G,VertexType u)
{ // 初始条件：图 G 存在,u 和 G 中顶点有相同特征(顶点名称相同)
  // 操作结果：若 G 中存在顶点 u,则返回该顶点在图中位置(序号)；否则返回 -1
  int i;
  for(i = 0;i<G.vexnum;++i) // 对于所有顶点依次查找
    if(strcmp(u.name,G.vertices[i].data.name) == 0) // 顶点与给定的 u 的顶点名称相同
      return i; // 返回顶点序号
  return -1; // 图 G 中不存在与顶点 u 有相同名称的顶点
}
void CreateGraph(ALGraph &G)
{ // 采用邻接表存储结构,构造图或网 G(用一个函数构造 4 种图)
  int i,j,k;
  VertexType v1,v2; // 顶点类型
  ElemType e; // 表结点的元素类型(存储弧的信息)
  char s[3] = "边";
  printf("请输入图的类型(有向图:0 有向网:1 无向图:2 无向网:3):");
  scanf("%d",&G.kind);
  if(G.kind<2) // 有向
    strcpy(s,"弧");
  printf("请输入图的顶点数,边数:");
  scanf("%d,%d",&G.vexnum,&G.arcnum);
  printf("请输入%d个顶点的值(名称<%d个字符):\n",G.vexnum,MAX_NAME);
  for(i = 0;i<G.vexnum;++i) // 构造顶点向量
  { Input(G.vertices[i].data); // 输入顶点信息
    G.vertices[i].firstarc = NULL; // 初始化与该顶点有关的出弧链表
  }
  printf("请输入%d条%s的",G.arcnum,s);
  switch(G.kind)
  { case  DG:printf("弧尾 弧头:\n"); // 设图没有弧(边)信息
             break;
    case  DN:printf("弧尾 弧头 弧的信息:\n");
             break;
    case UDG:printf("顶点1 顶点2:\n"); // 设图没有弧(边)信息
             break;
    case UDN:printf("顶点1 顶点2 边的信息:\n");
  }
```

```
    for(k = 0;k<G.arcnum;++k) // 构造相关弧链表
    { scanf("%s%s",v1.name,v2.name); // 输入 2 顶点名称
      i = LocateVex(G,v1); // 弧尾
      j = LocateVex(G,v2); // 弧头
      e.info = NULL; // 给待插表结点 e 赋值,设图无弧(边)信息
      if(G.kind%2) // 网
        InputArc(e.info); // 动态生成存储空间,输入弧的相关信息,在 func7-4.cpp 中
      e.adjvex = j; // 弧头
      ListInsert(G.vertices[i].firstarc,1,e);
      // 将 e 插在第 i 个元素(出弧)的表头,在 bo2-3.cpp 中
      if(G.kind>= 2) // 无向图或网,产生第 2 个表结点,并插在第 j 个元素(入弧)的表头
      { e.adjvex = i; // e.info 不变,不必再赋值
        ListInsert(G.vertices[j].firstarc,1,e); // 插在第 j 个元素的表头,在 bo2-3.cpp 中
      }
    }
}
void CreateFromFile(ALGraph &G,char* filename)
{ // 采用邻接表存储结构,由文件构造图或网 G(用一个函数构造 4 种图)
    int i,j,k;
    VertexType v1,v2; // 顶点类型
    ElemType e; // 表结点的元素类型(存储弧的信息)
    FILE *f; // 文件指针类型
    f = fopen(filename,"r"); // 以读的方式打开数据文件,并以 f 表示
    fscanf(f,"%d",&G.kind); // 由文件输入 G 的类型
    fscanf(f,"%d",&G.vexnum); // 由文件输入 G 的顶点数
    for(i = 0;i<G.vexnum;++i) // 构造顶点向量
    { InputFromFile(f,G.vertices[i].data); // 由文件输入顶点信息
      G.vertices[i].firstarc = NULL; // 初始化与该顶点有关的出弧链表
    }
    fscanf(f,"%d",&G.arcnum); // 由文件输入 G 的弧(边)数
    for(k = 0;k<G.arcnum;++k) // 构造相关弧链表
    { fscanf(f,"%s%s",v1.name,v2.name); // 由文件输入 2 顶点名称
      i = LocateVex(G,v1); // 弧尾
      j = LocateVex(G,v2); // 弧头
      e.info = NULL; // 给待插表结点 e 赋值,设图无弧(边)信息
      if(G.kind%2) // 网
        InputArcFromFile(f,e.info);
        // 动态生成存储空间,由文件输入弧的相关信息,在 func7-4.cpp 中
      e.adjvex = j; // 弧头
      ListInsert(G.vertices[i].firstarc,1,e);
      // 将 e 插在第 i 个元素(出弧)的表头,在 bo2-3.cpp 中
      if(G.kind>= 2) // 无向图或网,产生第 2 个表结点,并插在第 j 个元素(入弧)的表头
      { e.adjvex = i; // e.info 不变,不必再赋值
        ListInsert(G.vertices[j].firstarc,1,e); // 插在第 j 个元素的表头,在 bo2-3.cpp 中
      }
```

```
    }
    fclose(f); // 关闭数据文件
}
VertexType GetVex(ALGraph G,int v)
{ // 初始条件:图 G 存在,v 是 G 中某个顶点的序号。操作结果:返回 v 的值
    if(v>=G.vexnum||v<0) // 图 G 中不存在序号为 v 的顶点
        exit(OVERFLOW);
    return G.vertices[v].data; // 返回该顶点的信息
}
Status PutVex(ALGraph &G,VertexType v,VertexType value)
{ // 初始条件:图 G 存在,v 是 G 中某个顶点。操作结果:对 v 赋新值 value
    int k=LocateVex(G,v); // k 为顶点 v 在图 G 中的序号
    if(k!=-1) // v 是 G 的顶点
    { G.vertices[k].data=value; // 将新值赋给顶点 v(其序号为 k)
        return OK;
    }
    return ERROR; // v 不是 G 的顶点
}
int FirstAdjVex(ALGraph G,int v)
{ // 初始条件:图 G 存在,v 是 G 中某个顶点的序号
    // 操作结果:返回 v 的第 1 个邻接顶点的序号。若顶点在 G 中没有邻接顶点,则返回-1
    ArcNode *p=G.vertices[v].firstarc; // p 指向顶点 v 的第 1 个邻接顶点
    if(p) // 顶点 v 有邻接顶点
        return p->data.adjvex; // 返回 v 的第 1 个邻接顶点的序号
    else
        return -1; // 顶点 v 没有邻接顶点
}
Status equalvex(ElemType a,ElemType b)
{ // DeleteArc()、DeleteVex()和 NextAdjVex()要调用的函数
    if(a.adjvex==b.adjvex) // 表结点的顶位置(序号)相同
        return OK;
    else
        return ERROR;
}
int NextAdjVex(ALGraph G,int v,int w)
{ // 初始条件:图 G 存在,v 是 G 中某个顶点的序号,w 是 v 的邻接顶点的序号
  // 操作结果:返回 v 的(相对于 w 的)下一个邻接顶点的序号。
  //           若 w 是 v 的最后一个邻接顶点,则返回-1
    LinkList p,p1; // p1 在 Point()中用作辅助指针,Point()在 func2-4.cpp 中
    ElemType e; // 表结点的元素类型(存储弧的信息)
    e.adjvex=w;
    p=Point(G.vertices[v].firstarc,e,equalvex,p1);
    // p 指向顶点 v 的链表中邻接顶点为 w 的结点
    if(!p||!p->next) // 未找到 w 或 w 是最后一个邻接点
        return -1;
```

```
    else // p->data.adjvex == w
      return p->next->data.adjvex; // 返回 v 的(相对于 w 的)下一个邻接顶点的序号
}
void InsertVex(ALGraph &G,VertexType v)
{ // 初始条件：图 G 存在,v 和图中顶点有相同特征
  // 操作结果：在图 G 中增添新顶点 v(不增添与顶点相关的弧,留待 InsertArc()去做)
  G.vertices[G.vexnum].data = v; // 构造新顶点向量
  G.vertices[G.vexnum].firstarc = NULL; // 没有与顶点相关的弧
  G.vexnum++; // 图 G 的顶点数加 1
}
Status InsertArc(ALGraph &G,VertexType v,VertexType w)
{ // 初始条件：图 G 存在,v 和 w 是 G 中两个顶点
  // 操作结果：在 G 中增添弧<v,w>,若 G 是无向的,则还增添对称弧<w,v>
  ElemType e; // 表结点的元素类型(存储弧的信息)
  int i,j;
  char s1[3] = "边",s2[3] = "—"; // 无向的情况
  if(G.kind<2) // 有向
  { strcpy(s1,"弧");
    strcpy(s2,"→");
  }
  i = LocateVex(G,v); // 弧尾或边的序号
  j = LocateVex(G,w); // 弧头或边的序号
  if(i<0||j<0) // v 和 w 至少有 1 个不是 G 中的顶点
    return ERROR;
  G.arcnum++; // 图 G 的弧或边的数目加 1
  e.adjvex = j; // 弧头表结点的值
  e.info = NULL; // 初值,设图无弧(边)信息
  if(G.kind % 2) // 网
  { printf("请输入 %s%s%s%s 的信息：",s1,v.name,s2,w.name);
    InputArc(e.info); // 动态生成存储空间,输入弧的相关信息,在 func7-4.cpp 中
  }
  ListInsert(G.vertices[i].firstarc,1,e); // 将 e 插在弧尾的表头,在 bo2-3.cpp 中
  if(G.kind>= 2) // 无向,生成另一个表结点
  { e.adjvex = i; // 弧尾表结点的值,e.info 不变
    ListInsert(G.vertices[j].firstarc,1,e); // 将 e 插在弧头的表头
  }
  return OK;
}
Status DeleteArc(ALGraph &G,VertexType v,VertexType w)
{ // 初始条件：图 G 存在,v 和 w 是 G 中两个顶点
  // 操作结果：在 G 中删除弧<v,w>,若 G 是无向的,则还删除对称弧<w,v>
  int i,j,n;
  ElemType e; // 表结点的元素类型(存储弧的信息)
  i = LocateVex(G,v); // i 是顶点 v(弧尾)的序号
```

```
    j = LocateVex(G,w); // j 是顶点 w(弧头)的序号
    if(i<0||j<0||i == j) // v 和 w 至少有 1 个不是 G 中的顶点,或 v 和 w 是 G 中的同一个顶点
      return ERROR;
    e.adjvex = j; // 弧头表结点的值
    n = LocateElem(G.vertices[i].firstarc,e,equalvex);
    // 在弧尾链表中找弧头表结点,将其在链表中的位序赋给 n
    if(n) // 存在该弧
    { ListDelete(G.vertices[i].firstarc,n,e); // 在弧尾链表中删除弧头表结点,并用 e 返回其值
      G.arcnum--; // 弧或边数减 1
      if(G.kind%2) // 网,设图无弧(边)信息
        free(e.info); // 释放动态生成的弧(边)信息空间
      if(G.kind>= 2) // 无向,删除对称弧<w,v>
      { e.adjvex = i; // 弧尾表结点的值
        n = LocateElem(G.vertices[j].firstarc,e,equalvex);
        // 在弧头链表中找弧尾表结点,将其在链表中的位序赋给 n
        ListDelete(G.vertices[j].firstarc,n,e);
        // 在弧头链表中删除弧尾表结点,并用 e 返回其值
      }
      return OK;
    }
    else // 未找到待删除的弧
      return ERROR;
  }
  Status DeleteVex(ALGraph &G,VertexType v)
  { // 初始条件:图 G 存在,v 是 G 中某个顶点。操作结果:删除 G 中顶点 v 及其相关的弧(边)
    int i,k;
    LinkList p; // 表结点的指针类型
    k = LocateVex(G,v); // k 为待删除顶点 v 的序号
    if(k<0) // v 不是图 G 的顶点
      return ERROR;
    for(i = 0;i<G.vexnum;i++)
      DeleteArc(G,v,G.vertices[i].data); // 删除由顶点 v 发出的所有弧
    if(G.kind<2) // 有向
      for(i = 0;i<G.vexnum;i++)
        DeleteArc(G,G.vertices[i].data,v); // 删除发向顶点 v 的所有弧
    for(i = 0;i<G.vexnum;i++) // 对于 adjvex 域>k 的结点,其序号 - 1
    { p = G.vertices[i].firstarc; // p 指向弧结点的单链表
      while(p) // 未到表尾
      { if(p->data.adjvex>k) // adjvex 域>k
          p->data.adjvex--; // 序号 - 1(因为前移)
        p = p->next; // p 指向下一个结点
      }
    }
    for(i = k + 1;i<G.vexnum;i++)
```

```
    G.vertices[i-1]=G.vertices[i]; // 顶点 v 后面的顶点依次前移
  G.vexnum--; // 顶点数减 1
  return OK;
}
void DestroyGraph(ALGraph &G)
{ // 初始条件：图 G 存在。操作结果：销毁图 G
  int i;
  for(i=G.vexnum-1;i>=0;i--) // 由大到小逐一删除顶点及与其相关的弧(边)
    DeleteVex(G,G.vertices[i].data);
}
void Display(ALGraph G)
{ // 输出图的邻接矩阵 G
  int i;
  ArcNode *p;
  char s1[3]="边",s2[3]="—"; // 无向的情况
  if(G.kind<2) // 有向
  { strcpy(s1,"弧");
    strcpy(s2,"→");
  }
  switch(G.kind)
  { case  DG:printf("有向图\n");
             break;
    case  DN:printf("有向网\n");
             break;
    case UDG:printf("无向图\n");
             break;
    case UDN:printf("无向网\n");
  }
  printf("%d 个顶点,依次是：",G.vexnum);
  for(i=0;i<G.vexnum;++i)
    Visit(GetVex(G,i)); // 根据顶点信息的类型,访问第 i 个顶点,在 func7-1.cpp 中
  printf("\n%d 条%s：\n",G.arcnum,s1);
  for(i=0;i<G.vexnum;i++)
  { p=G.vertices[i].firstarc; // p 指向序号为 i 的顶点的第 1 条弧(边)
    while(p) // p 不为空
    { if(G.kind<=1||i<p->data.adjvex) // 有向或无向两次中的一次
      { printf("%s%s%s",G.vertices[i].data.name,s2,
        G.vertices[p->data.adjvex].data.name);
        if(G.kind%2) // 网
          OutputArc(p->data.info); // 输出弧(边)信息(包括权值),在 func7-4.cpp 中
      }
      p=p->nextarc; // p 指向下一个表结点
    }
    printf("\n");
```

```
  }
}
```

```
// main7-2.cpp 检验 bo7-2.cpp 的主程序
#include"c1.h"
#include"func7-1.cpp" // 包括顶点信息类型的定义及对它的操作
#include"func7-4.cpp" // 弧(边)的相关信息类型的定义及对它的操作
#include"c7-2'.h" // 图的邻接表存储结构(与单链表的变量类型建立联系)
#include"bo7-2.cpp" // 图的邻接表存储结构的基本操作(14 个)
typedef ALGraph Graph; // 定义 func7-3.cpp 中 Graph 的类型为 ALGraph(邻接表存储结构)
#include"func7-3.cpp" // 主函数(和 main7-1.cpp 共用)
```

**程序运行结果：**

```
请依次选择有向图,有向网,无向图,无向网:
请输入图的类型(有向图:0 有向网:1 无向图:2 无向网:3):0↙
请输入图的顶点数,边数:2,1↙
请输入 2 个顶点的值(名称<9 个字符):
a1 a2↙
请输入 1 条弧的弧尾 弧头:
a2 a1↙
有向图(见图 7-22)
2 个顶点,依次是:a1 a2
1 条弧:

a2→a1
插入新顶点,请输入新顶点的值:a3↙
插入与新顶点有关的弧,请输入弧数:2↙
请输入另一顶点的名称:a1↙
请输入另一顶点的方向(0:弧头 1:弧尾):0↙
请输入另一顶点的名称:a2↙
请输入另一顶点的方向(0:弧头 1:弧尾):1↙
有向图(见图 7-23)
3 个顶点,依次是:a1 a2 a3
3 条弧:

a2→a3 a2→a1
a3→a1
删除顶点及相关的弧,请输入待删除顶点的名称:a1↙
有向图(见图 7-24)
2 个顶点,依次是:a2 a3
1 条弧:
a2→a3
请输入图的类型(有向图:0 有向网:1 无向图:2 无向网:3):1↙
请输入图的顶点数,边数:2,1↙
```



图 7-22　有向图



图 7-23　插入顶点 a3



图 7-24　删除顶点 a1

```
请输入 2 个顶点的值(名称<9 个字符):
b1 b2 ↙
请输入 1 条弧的弧尾 弧头 弧的信息:
b1 b2 3 ↙
有向网(见图 7-25)
2 个顶点,依次是:b1 b2
1 条弧:
b1→b2:3
```

图 7-25 有向网

```
插入新顶点,请输入新顶点的值:b3 ↙
插入与新顶点有关的弧,请输入弧数:2 ↙
请输入另一顶点的名称:b1 ↙
请输入另一顶点的方向(0:弧头 1:弧尾):0 ↙
请输入弧 b3→b1 的信息:5 ↙
请输入另一顶点的名称:b2 ↙
请输入另一顶点的方向(0:弧头 1:弧尾):1 ↙
请输入弧 b2→b3 的信息:6 ↙
有向网(见图 7-26)
3 个顶点,依次是:b1 b2 b3
3 条弧:
b1→b2:3
b2→b3:6
b3→b1:5
```

图 7-26 插入顶点 b3

```
删除顶点及相关的弧,请输入待删除顶点的名称:b2 ↙
有向网(见图 7-27)
2 个顶点,依次是:b1 b3
1 条弧:

b3→b1:5
```

图 7-27 删除顶点 b2

```
请输入图的类型(有向图:0 有向网:1 无向图:2 无向网:3):2 ↙
请输入图的顶点数,边数:2,1 ↙
请输入 2 个顶点的值(名称<9 个字符):
c1 c2 ↙
请输入 1 条边的顶点 1 顶点 2:
c1 c2 ↙
无向图(见图 7-28)
2 个顶点,依次是:c1 c2
1 条边:
c1—c2
```

图 7-28 无向图

```
插入新顶点,请输入新顶点的值:c3 ↙
插入与新顶点有关的弧,请输入弧数:2 ↙
请输入另一顶点的名称:c1 ↙
请输入另一顶点的名称:c2 ↙
```

无向图(见图 7-29)
3 个顶点,依次是：c1 c2 c3
3 条边：
c1—c3 c1—c2
c2—c3

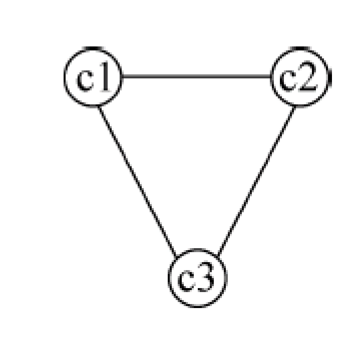


图 7-29　插入顶点 c3


删除顶点及相关的弧,请输入待删除顶点的名称：c3↙
无向图(见图 7-30)
2 个顶点,依次是：c1 c2
1 条边：
c1—c2



图 7-30　删除顶点 c3

请输入图的类型(有向图：0 有向网：1 无向图：2 无向网：3)：3↙
请输入图的顶点数,边数：2,1↙
请输入 2 个顶点的值(名称<9 个字符)：
d1 d2↙
请输入 1 条边的顶点 1 顶点 2 边的信息：
d1 d2 5↙
无向网(见图 7-31)
2 个顶点,依次是：d1 d2
1 条边：
d1—d2：5

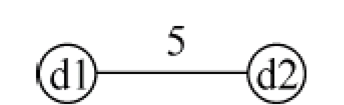


图 7-31　无向网

插入新顶点,请输入新顶点的值：d3↙
插入与新顶点有关的弧,请输入弧数：2↙
请输入另一顶点的名称：d1↙
请输入边 d3—d1 的信息：4↙
请输入另一顶点的名称：d2↙
请输入边 d3—d2 的信息：6↙
无向网(见图 7-32)
3 个顶点,依次是：d1 d2 d3
3 条边：
d1—d3：4 d1—d2：5
d2—d3：6

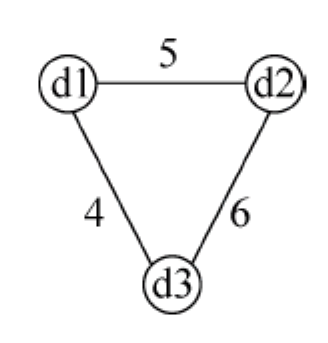


图 7-32　插入顶点 d3

删除顶点及相关的弧,请输入待删除顶点的名称：d2↙
无向网(见图 7-33)
2 个顶点,依次是：d1 d3
1 条边：
d1—d3：4

图 7-33　删除顶点 d2

修改顶点的值,请输入待修改顶点名称 新值：d1 D1↙
删除一条弧,请输入待删除弧的顶点 1 顶点 2：D1 d3↙
无向网(见图 7-34)
2 个顶点,依次是：D1 d3
0 条边：

图 7-34　删除边 D1—d3

# 7.2 图的遍历

对图的搜索就是对图中顶点的遍历。图中各顶点的关系比较复杂、一个顶点可能有多个邻接顶点，也可能是独立顶点(非连通图)。为了不重复地访问所有顶点，需要为顶点向量设立一个访问标志数组 visite[]，并置其初值为 FALSE(未被访问)。遍历时只访问那些未被访问过的顶点，且在访问后，将其访问标志的值改为 TRUE。当所有顶点访问标志的值都为 TRUE，则图已完成遍历。遍历一般从图的第 1 个顶点开始。确定遍历顶点的顺序有两个搜索原则：深度优先搜索和广度优先搜索。

## 7.2.1 深度优先搜索

```
// bo7-3.cpp 算法 7.4～算法 7.6
Boolean visite[MAX_VERTEX_NUM]; // 访问标志数组(全局量)
void( * VisitFunc)(VertexType); // 函数变量
void DFS(Graph G,int v)
{ // 从第 v 个顶点出发递归地深度优先遍历图 G。算法 7.5
  int w;
  visite[v] = TRUE; // 设置访问标志为 TRUE(已访问)
  VisitFunc(GetVex(G,v)); // 访问第 v 个顶点
  for(w = FirstAdjVex(G,v);w>= 0;w = NextAdjVex(G,v,w)) // 从顶点 v 的第 1 个邻接顶点 w 开始
    if(!visite[w]) // 邻接顶点 w 尚未被访问
      DFS(G,w); // 对 v 的尚未访问的序号为 w 的邻接顶点递归调用 DFS(访问 w)
}
void DFSTraverse(Graph G,void(* Visit)(VertexType))
{ // 初始条件：图 G 存在，Visit 是顶点的应用函数。算法 7.4
  // 操作结果：从第 1 个顶点起，深度优先遍历图 G，并对每个顶点调用函数 Visit 一次且仅一次
  int v;
  VisitFunc = Visit; // 使用全局变量 VisitFunc，使 DFS 不必设函数指针参数
  for(v = 0;v<G.vexnum;v++ ) // 对图 G 的所有顶点
    visite[v] = FALSE; // 访问标志数组初始化(未被访问)
  for(v = 0;v<G.vexnum;v++ ) // 对图 G 的所有顶点
    if(!visite[v]) // 顶点 v 尚未被访问
      DFS(G,v); // 对尚未访问的序号为 v 的顶点调用 DFS
  printf("\n");
}
typedef int QElemType; // 定义队列元素类型为整型(存储顶点序号)
#include"c3-2.h" // 链队列的结构，BFSTraverse()用
#include"bo3-2.cpp" // 链队列的基本操作，BFSTraverse()用
void BFSTraverse(Graph G,void(* Visit)(VertexType))
{ // 初始条件：图 G 存在，Visit 是顶点的应用函数。算法 7.6
  // 操作结果：从第 1 个顶点起，按广度优先非递归遍历图 G，
  //           并对每个顶点调用函数 Visit 一次且仅一次
```

```
  int v,u,w;
  LinkQueue Q; // 使用辅助队列 Q 和访问标志数组 visite
  for(v=0;v<G.vexnum;v++) // 对图 G 的所有顶点
    visite[v]=FALSE; // 访问标志数组初始化(未被访问)
  InitQueue(Q); // 初始化辅助队列 Q
  for(v=0;v<G.vexnum;v++) // 对图 G 的所有顶点
    if(!visite[v]) // 顶点 v 尚未被访问
    { visite[v]=TRUE; // 设置访问标志为 TRUE(已访问)
      Visit(GetVex(G,v)); // 访问顶点 v,在 func7-1.cpp 中
      EnQueue(Q,v); // v 入队列 Q
      while(!QueueEmpty(Q)) // 队列 Q 不空
      { DeQueue(Q,u); // 队头元素出队并置为 u
        for(w=FirstAdjVex(G,u);w>=0;w=NextAdjVex(G,u,w)) // 从 u 的第 1 个邻接顶点 w 起
          if(!visite[w]) // w 为 u 的尚未访问的邻接顶点
          { visite[w]=TRUE; // 设置访问标志为 TRUE(已访问)
            Visit(GetVex(G,w)); // 访问顶点 w
            EnQueue(Q,w); // w 入队列 Q
          }
      }
    }
  printf("\n");
}
```

算法 7.4 和算法 7.5(在 bo7-3.cpp 中)是利用递归对图进行深度优先搜索遍历的算法,它的主要思想是:在访问一个顶点(假设为 A)之后,不是马上依照顶点的存储顺序访问下一个顶点(假设为 B),而是先依次访问顶点 A 的所有邻接顶点。同样,在访问了顶点 A 的第 1 个邻接顶点(假设为 C)之后,又去先访问顶点 C 的所有邻接顶点。所以,对顶点 B(假设它与顶点 A 不连通)的访问是在对与顶点 A 连通的所有顶点的访问之后进行的。所谓第 1 个邻接顶点、第 2 个邻接顶点不是由图的拓扑关系决定的,它取决于图的存储结构。即使是同一个图,如果它的存储结构不同,那么它的某个顶点的第 1 个邻接顶点、第 2 个邻接顶点也可能不同。关于这一点,将在后面 algo7-1.cpp 中根据实例作进一步的说明。

### 7.2.2　广度优先搜索

算法 7.6(在 bo7-3.cpp 中)是对图进行广度优先搜索遍历的算法,它的主要思想是:先访问图的第 1 个顶点,然后依次访问这个顶点的所有邻接顶点,再依次访问这些邻接顶点的所有邻接顶点。这需要建立 1 个先进先出的队列,依次将访问过的顶点入队。当前一个顶点的所有邻接顶点都被访问了,就出队 1 个顶点,再访问这个顶点的所有邻接顶点并将这些邻接顶点入队。直至所有顶点都被访问过。

算法 7.4、算法 7.5 和算法 7.6 是基于基本操作的,所以可用于多种图的存储结构。algo7-1.cpp 是在图的 2 种存储结构下,调用算法 7.4、算法 7.5 和算法 7.6,对图进行深度优先搜索遍历和广度优先搜索遍历的程序。

```
// algo7-1.cpp 检验 2 种结构调用算法 7.4～算法 7.6(深度优先和广度优先)搜索遍历的程序
// 如果采用 ALGraph 类型为图的存储结构,将下行作为注释
#define MG // 图的存储结构为 MGraph。第 3 行
#include"c1.h"
#include"func7-1.cpp" // 包括顶点信息类型的定义及对它的操作
#ifdef MG // 图的存储结构为 MGraph
  #include"func7-2.cpp" // 包括弧(边)的相关信息类型的定义及对它的操作
  #include"c7-1.h" // 图的数组(邻接矩阵)存储结构
  #include"bo7-1.cpp" // 图的数组(邻接矩阵)存储结构的基本操作
  typedef MGraph Graph; // 定义图的存储结构为邻接矩阵
#else // 图的存储结构为 ALGraph
  #include"func7-4.cpp" // 弧(边)的相关信息类型的定义及对它的操作
  #include"c7-2'.h" // 图的邻接表存储结构(与单链表的变量类型建立联系)
  #include"bo7-2.cpp" // 邻接表存储结构的基本操作
  typedef ALGraph Graph; // 定义图的存储结构为邻接表
#endif
#include"bo7-3.cpp" // 算法 7.4～算法 7.6
void main()
{
  Graph g; // 抽象的图类型 g
  char filename[13]; // 存储数据文件名(包括路径)
  printf("请输入数据文件名:");
  scanf("%s",filename);
#ifdef MG // 图的数组(邻接矩阵)存储结构
  CreateFromFile(g,filename,0); // 创建无相关信息的图
#else // 图的邻接表存储结构
  CreateFromFile(g,filename); // 创建无相关信息的图
#endif
  printf("深度优先搜索遍历的结果:\n");
  DFSTraverse(g,Visit);
  printf("广度优先搜索遍历的结果:\n");
  BFSTraverse(g,Visit);
}
```



图 7-35 数据文件 f7-1.txt 所表示的无向图

数据文件 f7-1.txt 的内容(图 7-35 是其所表示的无向图):

```
2
8
a b c d e f g h
14
a b
a c
a e
a f
a g
a h
```

```
b d
b e
b h
c g
c h
d h
e f
f g
```

数据文件 f7-2. txt 所表示的无向图与 f7-1. txt 的一样，也如图 7-35 所示。只是边的输入顺序不同。

数据文件 f7-2. txt 的内容：

```
2
8
a b c d e f g h
14
f g
e f
d h
c h
c g
b h
b e
b d
a h
a g
a f
a e
a c
a b
```

**程序运行结果：**

```
请输入数据文件名：f7-1. txt↙
深度优先搜索遍历的结果：
a b d h c g f e
广度优先搜索遍历的结果：
a b c e f g h d
```

如果用数据文件 f7-2. txt 取代 f7-1. txt，程序运行结果也是一样的。因为两文件创建的邻接矩阵是一样的。

如果将 algo7-1. cpp 的第 3 行宏定义“#define MG”作为注释行，则采用了邻接表存储结构。程序运行结果：

```
请输入数据文件名：f7-1.txt↙
深度优先搜索遍历的结果：
a h d b e f g c
广度优先搜索遍历的结果：
a h g f e c b d
```

如果用数据文件 f7-2.txt 取代 f7-1.txt，程序运行结果：

```
请输入数据文件名：f7-2.txt↙
深度优先搜索遍历的结果：
a b d h c g f e
广度优先搜索遍历的结果：
a b c e f g h d
```

注意到两数据文件产生的搜索遍历结果是不同的。原因是，虽然两文件创建的图的拓扑结构是一样的，但由于邻接表存储结构中边或弧是以链表形式存储的，且边或弧总是插在表头。当边或弧的输入顺序不同时，所创建的邻接表是不一样的，故搜索的顺序也会不同。f7-1.txt 创建的邻接表如图 7-36 所示（略去相关信息指针域，且为直观，用顶点名称代替顶点序号）；f7-2.txt 创建的邻接表如图 7-37 所示。当然，对于任一种图的存储结构，如果顶点的输入顺序不同，搜索的顺序也不同。

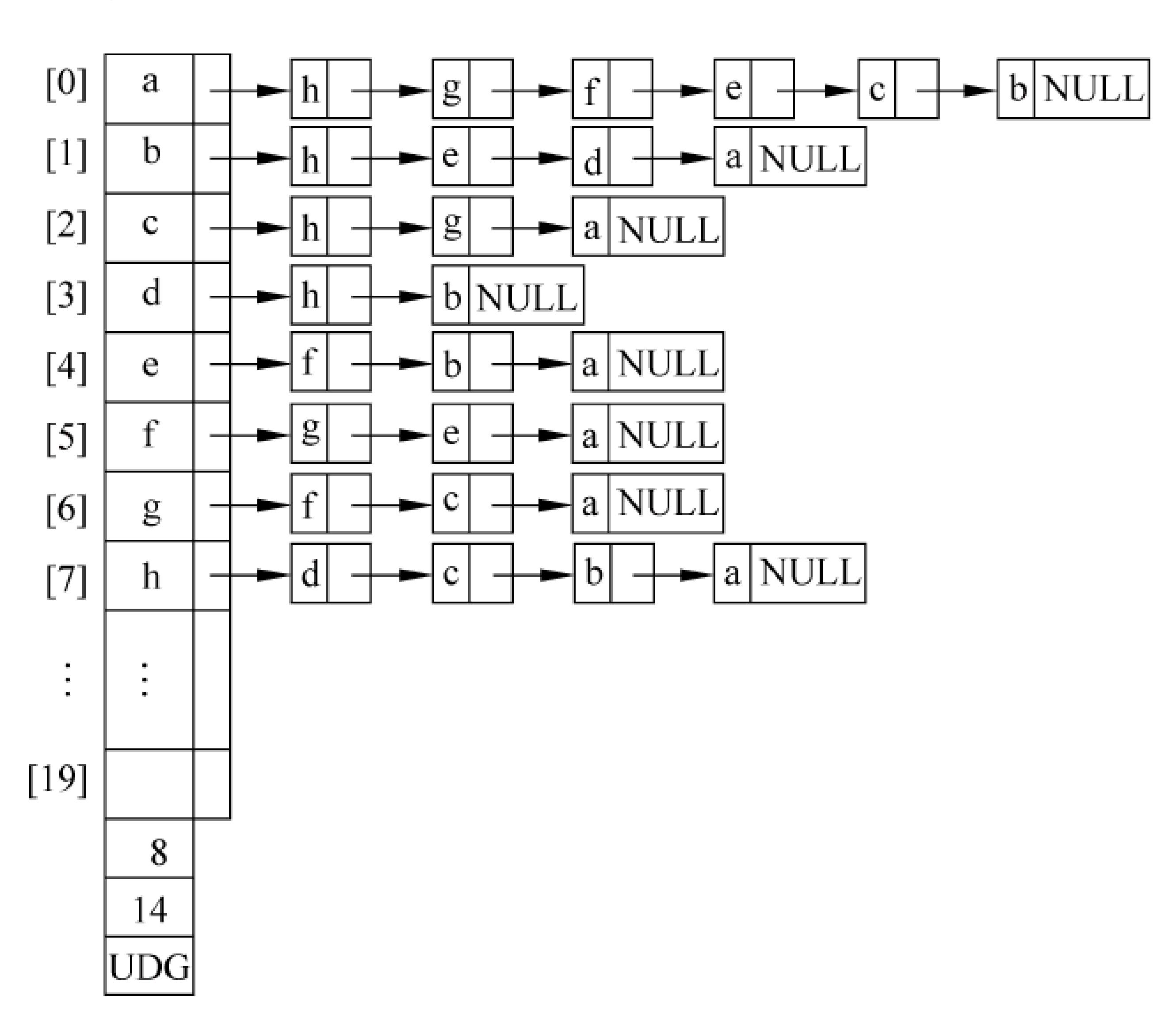



图 7-36　根据数据文件 f7-1.txt 所产生的邻接表

图有 2 个基本操作：FirstAdjVex(G,v)和 NextAdjVex(G,v,w)。FirstAdjVex(G,v)返回图 G 中序号为 v 的顶点的第 1 个邻接顶点的序号。NextAdjVex(G,v,w)比 FirstAdjVex(G,v)多了 1 个形参 w，它返回图 G 中序号为 v 的顶点的所有邻接顶点中排在序号为 w 的邻接顶点后面的那个邻接顶点的序号。

在邻接矩阵存储结构中，FirstAdjVex(G,v)返回邻接矩阵 G.arcs.adj 中序号为 v 的顶点所对应的行的第 1 个值为 1(图)或权值(网)的顶点的序号。以 f7-1.txt 所表示的无向图(如图 7-35 所示，设为图 G)为例，FirstAdjVex(G,0)(在 bo7-1.cpp 中)返回 a 的第 1 个邻接顶

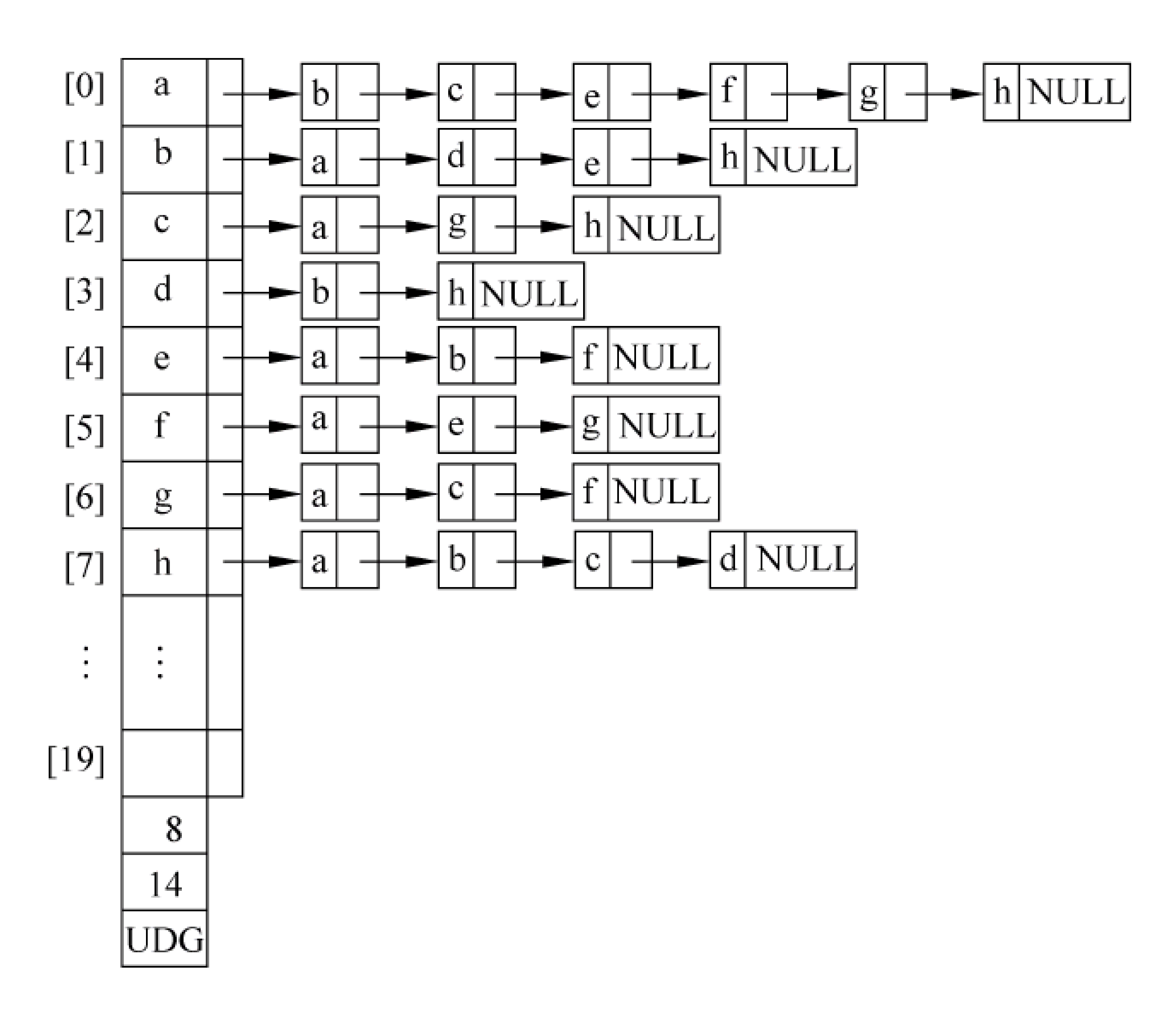



图 7-37　根据数据文件 f7-2. txt 所产生的邻接表

点 b 的序号 1；FirstAdjVex(G,4)返回 e 的第 1 个邻接顶点 a 的序号 0。NextAdjVex(G,v,w)返回邻接矩阵 G. arcs. adj 中序号为 v 的顶点所对应的行的序号为 w 那列后面的第 1 个值为 1(图)或权值(网)的顶点的序号。仍以 f7-1. txt 所表示的图 G 为例，NextAdjVex(G,0,2)(在 bo7-1. cpp 中)返回第 1 行(a 行)排在第 3 列(c 列)后面的第 1 个值为 1 的顶点 e 的序号 4。由这样的定义，我们可以推知，algo7-1. cpp(采用邻接矩阵存储结构)调用算法 7. 4 和算法 7. 5 对图 G 深度优先搜索遍历的过程：首先访问 G 的第 1 个顶点 a；接下来访问 a 的第 1 个邻接顶点 b；再准备访问 b 的第 1 个邻接顶点 a，但 a 已被访问过，则不再访问 a，转而访问 b 排在 a 后的邻接顶点 d；再准备访问 d 的第 1 个邻接顶点 b，由于同样的原因，转而访问 b 排在 d 后的邻接顶点 h；再访问 h 的第 1 个未被访问的邻接顶点 c、c 的第 1 个未被访问的邻接顶点 g、g 的第 1 个未被访问的邻接顶点 f、f 的第 1 个未被访问的邻接顶点 e。遍历结束，其顺序与程序运行结果相同。

algo7-1. cpp(采用邻接矩阵存储结构)调用算法 7. 6 对图 G 广度优先搜索遍历的过程：首先访问 G 的第 1 个顶点 a，将 a 入队，在队不空的情况下，出队元素 a，依次访问 a 的所有邻接顶点 b、c、e、f、g、h，并将其入队；出队 b，访问 b 的邻接顶点 d。遍历结束，其顺序与程序运行结果相同。

在邻接表存储结构中，FirstAdjVex(G,v)返回图 G 中序号为 v 的顶点的 firstarc 成员所指向的结点的序号。NextAdjVex(G,v,w)返回图 G 中序号为 v 的顶点的 firstarc 成员作为头指针的链表中排在序号为 w 的邻接顶点后面的那个邻接顶点的序号。以 f7-1. txt 创建的无向图存储结构(如图 7-36 所示，设为图 G)为例，FirstAdjVex(G,0)(在 bo7-2. cpp 中)返回 a 的第 1 个邻接顶点 h 的序号 7。NextAdjVex(G,0,2)(在 bo7-2. cpp 中)返回顶点 a 的 firstarc 链表中顶点 c 的直接后继顶点 b 的序号 1。

algo7-2. cpp 将算法 7. 4、算法 7. 5 和算法 7. 6 做了两点修改：

(1) 在函数 DFS()中不使用全局变量 VisitFunc，设置了函数指针参数 void( * Visit)(VertexType)，这使得 DFS()的形参多了 1 个；

(2) 将算法改为仅适用于邻接表存储结构,这样就可以直接用表结点的指针 p,从顶点的 firstarc 成员出发寻找其邻接顶点。用 p=G. vertices[v]. firstarc 和 p=p->next 代替 FirstAdjVex()和 NextAdjVex()的作用。这样做效率高、直观、更好理解,但仅适用于邻接表存储结构。

```
// algo7-2.cpp 邻接表存储结构的深度优先和广度优先搜索遍历
#include"c1.h"
#include"func7-1.cpp" // 包括顶点信息类型的定义及对它的操作
#include"func7-4.cpp" // 弧(边)的相关信息类型的定义及对它的操作
#include"c7-2'.h" // 图的邻接表存储结构(与单链表的变量类型建立联系)
#include"bo7-2.cpp" // 图的邻接表存储结构的基本操作
Boolean visited[MAX_VERTEX_NUM]; // 访问标志数组(全局量)
void DFS(ALGraph G,int v,void( * Visit)(VertexType))
{ // 从第 v 个顶点出发递归地深度优先遍历图 G。修改算法 7.5,仅适用于邻接表存储结构
  ArcNode * p; // p 指向表结点
  visited[v] = TRUE; // 设置访问标志为 TRUE(已访问)
  Visit(G.vertices[v].data); // 访问第 v 个顶点
  for(p = G.vertices[v].firstarc;p;p = p->next) // p 依次指向 v 的邻接顶点
    if(!visited[p->data.adjvex]) // 如果该邻接顶点尚未被访问
      DFS(G,p->data.adjvex,Visit); // 对 v 的尚未访问的邻接点递归调用 DFS
}
void DFSTraverse(ALGraph G,void( * Visit)(VertexType))
{ // 对图 G 作深度优先遍历。DFS 设函数指针参数,修改算法 7.4,仅适用于邻接表存储结构
  int v;
  for(v = 0;v<G.vexnum;v++ ) // 对于所有顶点
    visited[v] = FALSE; // 访问标志数组初始化,置初值为未被访问
  for(v = 0;v<G.vexnum;v++ ) // 如果是连通图,只 v = 0 就遍历全图
    if(!visited[v]) // v 尚未被访问
      DFS(G,v,Visit); // 对 v 调用 DFS
  printf("\n");
}
typedef int QElemType; // 定义队列元素类型为整型(存储顶点序号)
#include"c3-2.h" // 链队列的结构,BFSTraverse()用
#include"bo3-2.cpp" // 链队列的基本操作,BFSTraverse()用
void BFSTraverse(ALGraph G,void(* Visit)(VertexType)) // 修改算法 7.6,仅适用于邻接表结构
{ // 按广度优先非递归遍历图 G。使用辅助队列 Q 和访问标志数组 visited
  int v,u;
  ArcNode * p; // 表结点指针类型
  LinkQueue Q; // 链队列类型
  for(v = 0;v<G.vexnum;++v) // 对于所有顶点
    visited[v] = FALSE; // 置初值为未被访问
  InitQueue(Q); // 初始化辅助队列 Q
  for(v = 0;v<G.vexnum;v++ ) // 如果是连通图,只 v = 0 就遍历全图
    if(!visited[v]) // v 尚未被访问
    { visited[v] = TRUE; // 设 v 为已被访问
```

```
        Visit(G.vertices[v].data); // 访问 v
        EnQueue(Q,v); // v 入队列 Q
        while(!QueueEmpty(Q)) // 队列 Q 不空
        { DeQueue(Q,u); // 队头元素出队并赋给 u
          for(p=G.vertices[u].firstarc;p;p=p->next) // p 依次指向 u 的邻接顶点
            if(!visited[p->data.adjvex]) // u 的邻接顶点尚未被访问
            { visited[p->data.adjvex]=TRUE; // 设该邻接顶点为已被访问
              Visit(G.vertices[p->data.adjvex].data); // 访问该邻接顶点
              EnQueue(Q,p->data.adjvex); // 入队该邻接顶点序号
            }
        }
      }
  printf("\n");
}
void main()
{
  ALGraph g;
  char filename[13]; // 存储数据文件名(包括路径)
  printf("请输入数据文件名:");
  scanf("%s",filename);
  CreateFromFile(g,filename); // 利用数据文件创建无相关信息的图
  printf("深度优先搜索遍历的结果:\n");
  DFSTraverse(g,Visit);
  printf("广度优先搜索遍历的结果:\n");
  BFSTraverse(g,Visit);
}
```

**程序运行结果:**

```
请输入数据文件名:f7-1.txt↙
深度优先搜索遍历的结果:
a h d b e f g c
广度优先搜索遍历的结果:
a h g f e c b d
```

对比可见,algo7-2.cpp 程序运行结果与 algo7-1.cpp 在邻接表存储结构下的运行结果是一样的。因为虽然使用的语句不同,但算法是一样的,数据文件也是同样的。

## 7.3 图的连通性问题

### 7.3.1 无向图的连通分量和生成树

具有 n 个顶点的无向连通图至少有 n－1 条边,如果只有 n－1 条边,则不会形成环,这样的图称为“生成树”。连通图可通过遍历来构造生成树,非连通图的每个连通分量可构造

一棵生成树，整个非连通图构造为生成森林。algo7-3.cpp 调用算法 7.7 和算法 7.8，将无向图构造为生成森林，并以孩子-兄弟二叉链表存储之。

```
// algo7-3.cpp 调用算法 7.7 和算法 7.8
#include"c1.h"
#include"func7-1.cpp" // 包括顶点信息类型的定义及对它的操作
#include"func7-4.cpp" // 弧(边)的相关信息类型的定义及对它的操作
typedef VertexType TElemType; // 定义树的元素类型为图的顶点类型
#include"c6-4.h" // 孩子-兄弟二叉链表存储结构
#include"bo6-6.cpp" // 孩子-兄弟二叉链表存储结构的先根遍历操作
#include"c7-2'.h" // 图的邻接表存储结构(与单链表的变量类型建立联系)
Boolean visited[MAX_VERTEX_NUM]; // 访问标志数组(全局量)
#include"bo7-2.cpp" // 图的邻接表的基本操作
typedef ALGraph Graph; // 定义图的存储结构为邻接表
void DFSTree(Graph G,int v,CSTree &T)
{ // 从第 v 个顶点出发深度优先遍历图 G,建立以 T 为根的生成树。算法 7.8
  Boolean first = TRUE; // 树 T 还没有第 1 个孩子结点的标志
  int w;
  CSTree p,q; // 孩子-兄弟二叉链表结点的指针类型
  visited[v] = TRUE; // 顶点 v 已被访问的标志
  for(w = FirstAdjVex(G,v);w>= 0;w = NextAdjVex(G,v,w)) // w 依次为 v 的邻接顶点
    if(!visited[w]) // 顶点 w 尚未被访问
    { p = (CSTree)malloc(sizeof(CSNode)); // 分配孩子结点给 p
      p->data = GetVex(G,w); // 将顶点 w 的值赋给孩子结点的 data 域
      p->firstchild = NULL; // 孩子结点的 firstchild 域和 nextsibling 域赋空
      p->nextsibling = NULL;
      if(first) // 顶点 w 是顶点 v 的第 1 个未被访问的邻接顶点
      { T->firstchild = p; // 顶点 w 是根的第 1 个孩子结点
        first = FALSE; // 树 T 有第 1 个孩子结点的标志
      }
      else // 顶点 w 是顶点 v 的其他未被访问的邻接顶点
        q->nextsibling = p; // 是上一邻接顶点的兄弟姐妹结点
                          // for 循环的第 1 次不通过此处,以后 q 已赋值
      q = p; // q 指向 p 所指结点
      DFSTree(G,w,q); // 从第 w 个顶点出发深度优先遍历图 G,建立子生成树 q
    }
}
void DFSForest(Graph G,CSTree &T)
{ // 建立无向图 G 的深度优先生成森林的(最左)孩子(下一个)兄弟链表 T。算法 7.7
  CSTree p,q; // 孩子-兄弟二叉链表结点的指针类型
  int v;
  T = NULL; // 初始化生成森林的根结点为空
  for(v = 0;v<G.vexnum;++v) // 对于所有顶点
```

```
    visited[v] = FALSE; // 赋初值
  for(v = 0;v<G.vexnum;++v) // 对所有顶点 v
    if(!visited[v]) // 第 v 个顶点不曾被访问
    { // 第 v 个顶点为新的生成树的根结点
      p = (CSTree)malloc(sizeof(CSNode)); // 动态生成根结点
      p->data = GetVex(G,v); // 给根结点赋值
      p->firstchild = NULL; // 结点的 firstchild 域和 nextsibling 域赋空
      p->nextsibling = NULL; // 以下将 p 所指结点插到树 T 中
      if(!T) // p 所指结点是第 1 棵生成树 T 的根结点
        T = p; // T 指向 p 所指结点
      else // p 是其他生成树的根(前一棵树根的“兄弟”)
        q->nextsibling = p; // for 循环的第 1 次,T = NULL,不通过此处,以后 q 已由下句赋值
      q = p; // q 指示当前生成树的根
      DFSTree(G,v,p); // 建立以 p 为根的生成树
    }
}
void main()
{
  Graph g;
  CSTree t;
  printf("请选择无向图\n");
  CreateGraph(g); // 构造无向图 g
  Display(g); // 输出无向图 g
  DFSForest(g,t); // 建立无向图 g 的深度优先生成森林的孩子-兄弟链表 t
  printf("先序遍历生成森林:\n");
  PreOrderTraverse(t,Visit); // 先序遍历生成森林的孩子-兄弟链表 t
  printf("\n");
}
```

**程序运行结果**(以教科书中图 7.3(a)的 G3 为例):

```
请选择无向图
请输入图的类型(有向图:0 有向网:1 无向图:2 无向网:3):2↙
请输入图的顶点数,边数:13,13↙(见图 7-38)
请输入 13 个顶点的值(名称<9 个字符):
A B C D E F G H I J K L M↙
请输入 13 条边的顶点 1 顶点 2:
A B↙
A C↙
A F↙
A L↙
B M↙
D E↙
G H↙
G I↙
G K↙
H K↙
```

图 7-38　非连通无向图

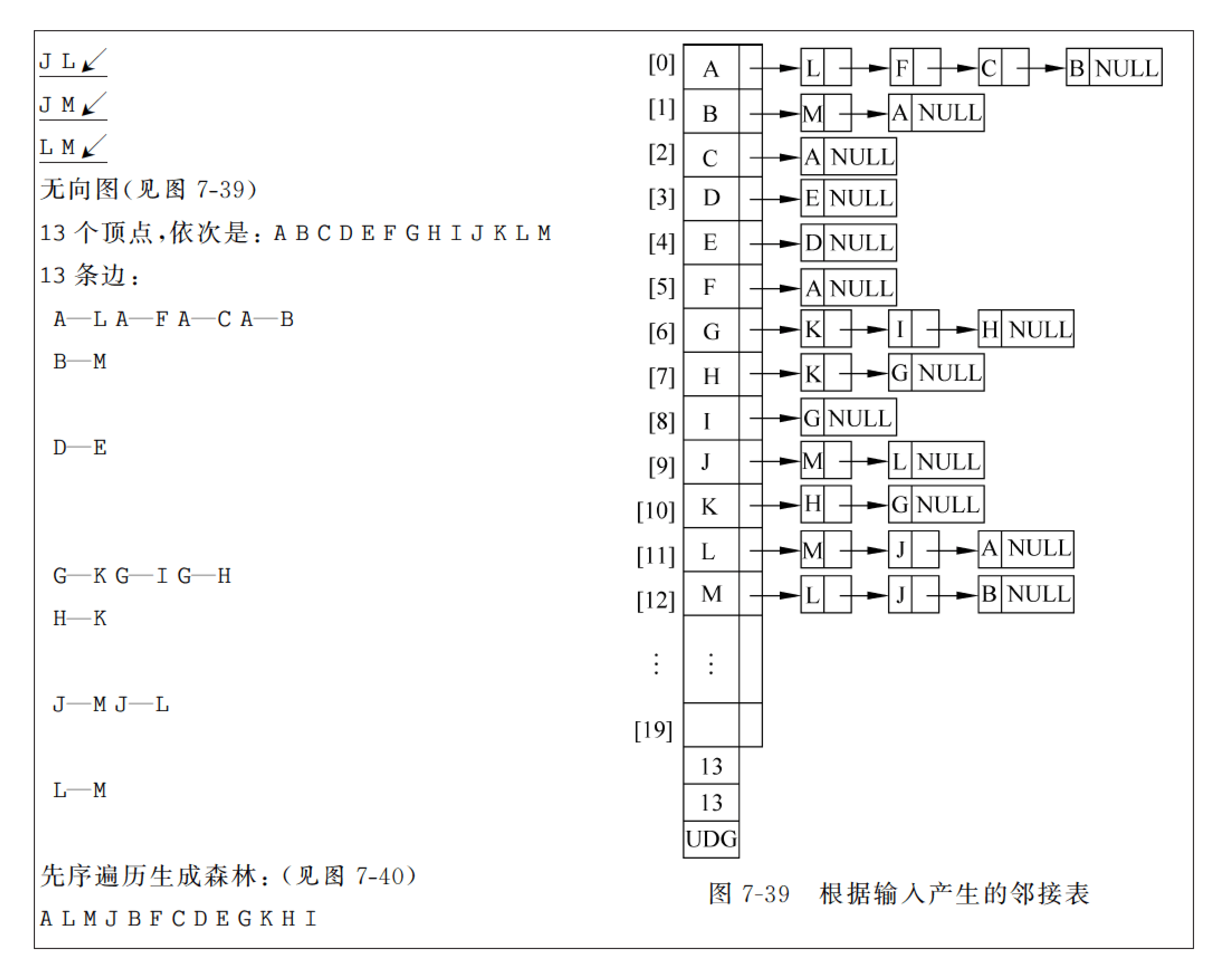


图 7-39 根据输入产生的邻接表

以上所输入的图的信息产生的邻接表如图 7-39 所示,仍略去相关信息指针域,并用顶点名称代替顶点位置。调用算法 7.7 产生的生成森林如图 7-40 所示,此生成森林以孩子-兄弟二叉链表存储的结构如图 7-41 所示。

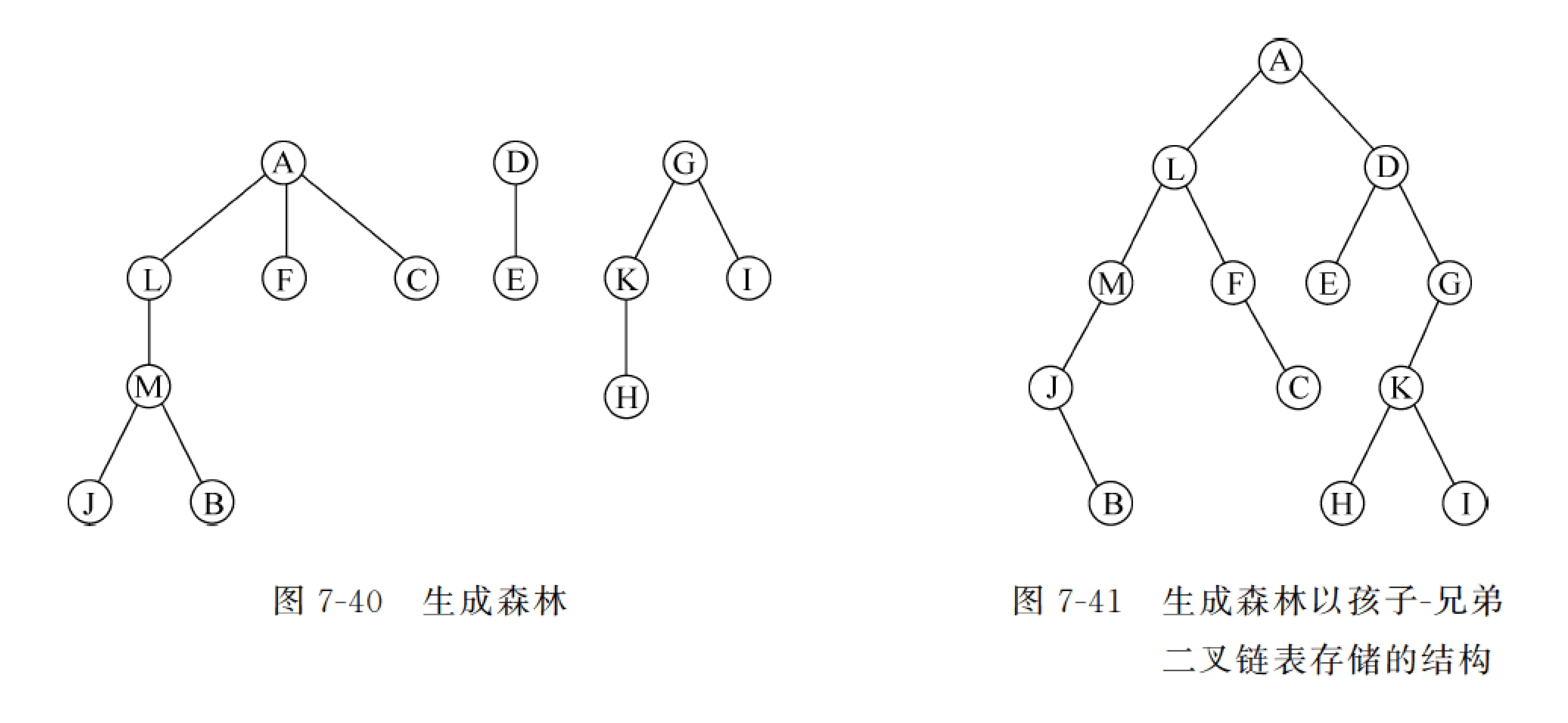

图 7-40 生成森林

图 7-41 生成森林以孩子-兄弟二叉链表存储的结构

算法 7.7 和算法 7.8 采用了抽象的图的存储类型 Graph,参考 algo7-1.cpp,很容易将 algo7-3.cpp改为在邻接矩阵存储结构下的应用。

## 7.3.2　最小生成树

```
// algo7-4.cpp 实现算法 7.9 的程序
#include"c1.h"
#include"func7-1.cpp" // 包括顶点信息类型的定义及对它的操作
#include"func7-2.cpp" // 包括弧(边)的相关信息类型的定义及对它的操作
#include"c7-1.h" // 图的数组(邻接矩阵)存储结构
#include"bo7-1.cpp" // 图的数组(邻接矩阵)存储结构的基本操作
typedef struct(见图 7-42)
{ // 记录从顶点集 U 到 V-U 的代价最小的边的辅助数组定义
  int adjvex; // 顶点集 U 中到该点为最小权值的那个顶点的序号
  VRType lowcost; // 那个顶点到该点的权值(最小权值)
}minside[MAX_VERTEX_NUM];
int minimum(minside SZ,MGraph G)
{ // 求 SZ.lowcost 的最小正值,并返回其在 SZ 中的序号
  int i=0,j,k,min;
  while(!SZ[i].lowcost) // 找第 1 个值不为 0 的 SZ[i].lowcost 的序号
    i++;
  min=SZ[i].lowcost; // min 标记第 1 个不为 0 的值
  k=i; // k 标记该值的序号
  for(j=i+1;j<G.vexnum;j++) // 继续向后找
    if(SZ[j].lowcost>0&&SZ[j].lowcost<min) // 找到新的更小的正值
    { min=SZ[j].lowcost; // min 标记此正值
      k=j; // k 标记此正值的序号
    }
  return k; // 返回当前最小正值在 SZ 中的序号
}
void MiniSpanTree_PRIM(MGraph G,VertexType u)
{ // 用普里姆算法从顶点 u 出发构造网 G 的最小生成树 T,输出 T 的各条边。算法 7.9
  int i,j,k;
  minside closedge;
  k=LocateVex(G,u); // 顶点 u 的序号
  for(j=0;j<G.vexnum;++j) // 辅助数组初始化
  { closedge[j].adjvex=k; // 顶点 u 的序号赋给 closedge[j].adjvex
    closedge[j].lowcost=G.arcs[k][j].adj; // 顶点 u 到该点的权值
  }
  closedge[k].lowcost=0; // 初始,U={u}。U 中的顶点到集合 U 的权值为 0
  printf("最小代价生成树的各条边为\n");
  for(i=1;i<G.vexnum;++i) // 选择其余 G.vexnum-1 个顶点(i 仅计数)
  { k=minimum(closedge,G); // 求出最小生成树 T 的下一个结点:第 k 顶点
    printf("(%s-%s)\n",G.vexs[closedge[k].adjvex].name,G.vexs[k].name);
    // 输出最小生成树 T 的边
    closedge[k].lowcost=0; // 第 k 顶点并入 U 集
    for(j=0;j<G.vexnum;++j)
      if(G.arcs[k][j].adj<closedge[j].lowcost) // 新顶点并入 U 集后重新选择最小边
```

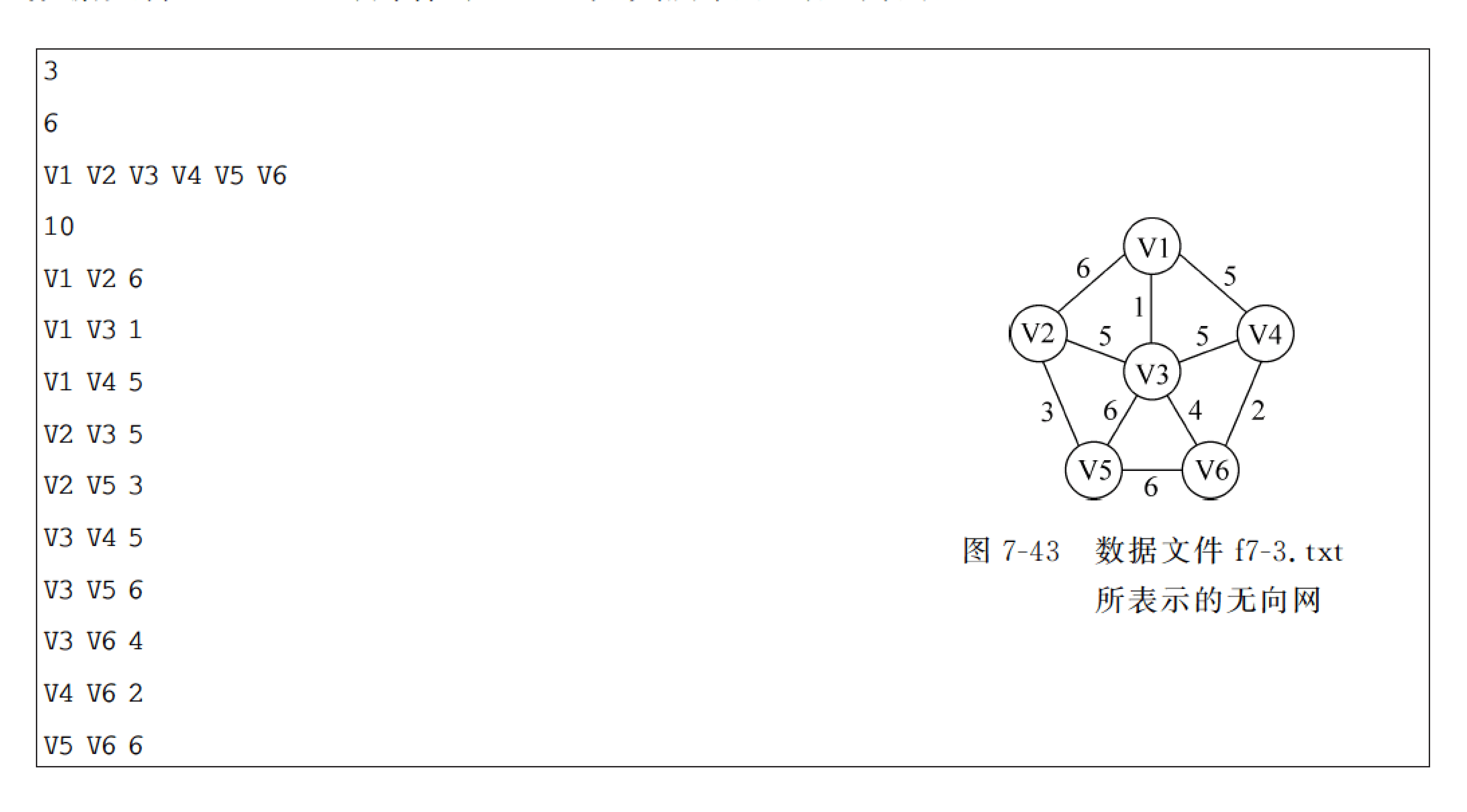
minside

| | | |
|---|---|---|
| [0] | adjvex | lowcost |
| [1] | | |
| [2] | | |
| ⋮ | ⋮ | |
| [25] | | |

图 7-42　minside 类型

```
        { closedge[j].adjvex = k;
          closedge[j].lowcost = G.arcs[k][j].adj;
        }
    }
}
void main()
{
  MGraph g;
  char filename[13]; // 存储数据文件名(包括路径)
  printf("请输入数据文件名:");
  scanf("%s",filename);
  CreateFromFile(g,filename,0); // 创建无相关信息的网
  Display(g); // 输出无向网 g
  MiniSpanTree_PRIM(g,g.vexs[0]);
  // 用普里姆算法从第 1 个顶点出发输出 g 的最小生成树的各条边
}
```

数据文件 f7-3.txt 的内容(图 7-43 是其所表示的无向网):



```
3
6
V1 V2 V3 V4 V5 V6
10
V1 V2 6
V1 V3 1
V1 V4 5
V2 V3 5
V2 V5 3
V3 V4 5
V3 V5 6
V3 V6 4
V4 V6 2
V5 V6 6
```



图 7-43 数据文件 f7-3.txt 所表示的无向网

**程序运行结果**(以教科书图 7.16 为例):

```
请输入数据文件名:f7-3.txt↙
6 个顶点 10 条边的无向网。顶点依次是:V1 V2 V3 V4 V5 V6 (见图 7-43)
G.arcs.adj:
      32767          6          1          5      32767      32767
          6      32767          5      32767          3      32767
          1          5      32767          5          6          4
          5      32767          5      32767      32767          2
      32767          3          6      32767      32767          6
      32767      32767          4          2          6      32767
```

```
G.arcs.info:
顶点 1 顶点 2 该边的信息:
最小代价生成树的各条边为
(V1 - V3)
(V3 - V6)
(V6 - V4)
(V3 - V2)
(V2 - V5)
```

图 7-44 是以上程序运行的说明，显示了算法 7.9(普里姆算法)求最小生成树的过程。首先，主程序构造了图 7-43 所示的无向网。然后，调用 MiniSpanTree_PRIM()，由顶点 V1 开始，求该网的最小生成树。最小生成树顶点集 U 最初只有 V1，其中用到了辅助数组 closedge[]。closedge[i].lowcost 是 U 中的顶点到顶点 i 的最小权值。若顶点 i 属于最小生成树，则 closedge[i].lowcost=0。closedge[i].adjvex 是最小生成树顶点集 U 中到顶点 i 为最小权值的那个顶点的序号。图 7-44(a)显示了 closedge[]的初态(为直观起见，图中用顶点名称代替顶点序号)。这时 U 中只有 V1，所以 closedge[i].adjvex 都是 V1，closedge[i].lowcost 是 V1 到顶点 i 的权值。closedge[0].lowcost=0，说明 V1 已属于 U 了。在 closedge[].lowcost 中找最小正数，closedge[2].lowcost=1，是最小正数。令 k=2，将 V3 并入 U(令 closedge[2].lowcost=0)，输出边(V1-V3)。因为 V3 到 V2、V5 和 V6 的权值小于 V1 到它们的权值，故将它们的 closedge[].lowcost 替换为 V3 到它们的权值；将它们的 closedge[].adjvex 替换为 V3，如图 7-44(b)所示。重复这个过程，依次如图 7-44(c)、(d)和(e)所示。最后，closedge[].adjvex 包含了最小生成树中每一条边的信息。

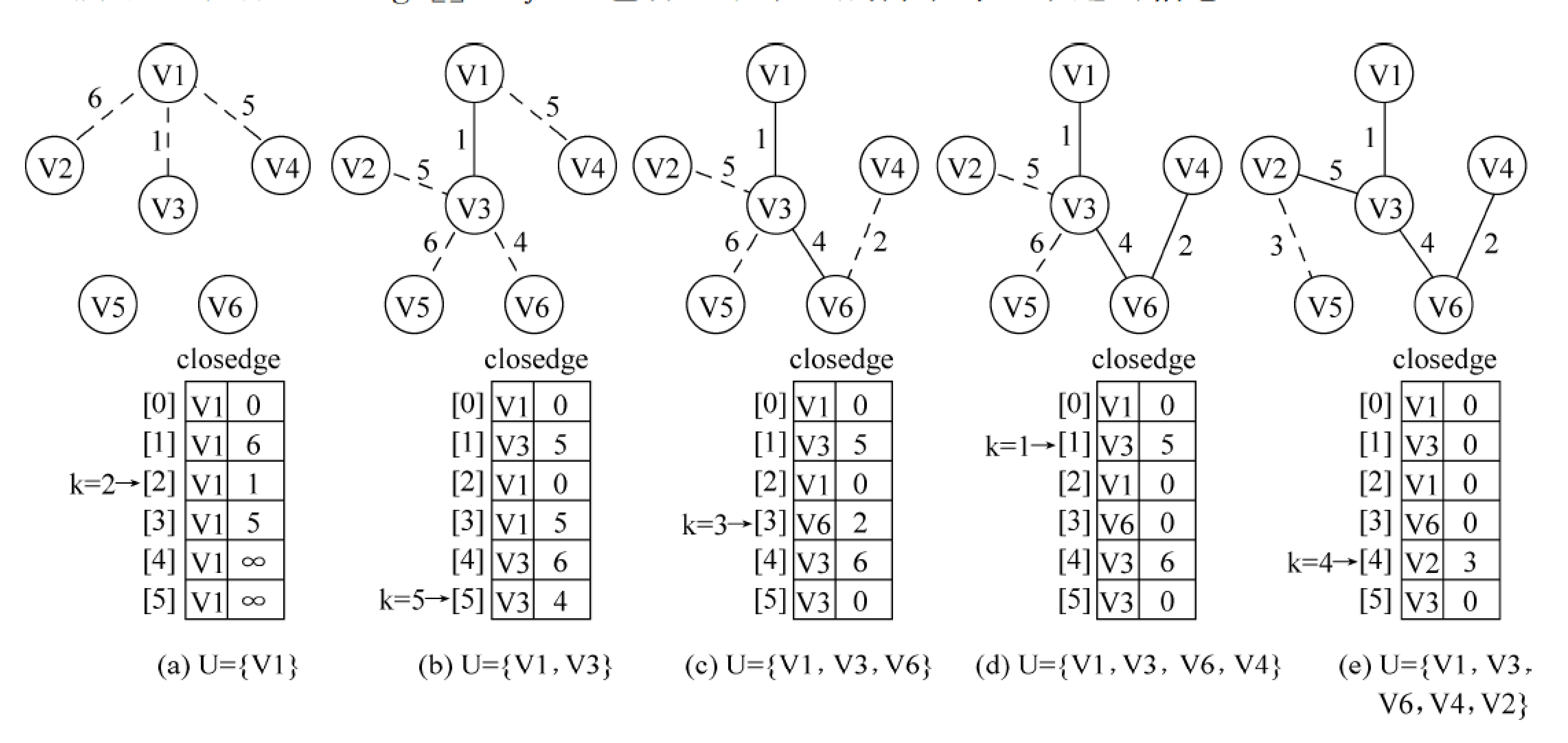



图 7-44　运行 algo7-4.cpp 过程

```
// algo7-5.cpp 克鲁斯卡尔算法求无向连通网的最小生成树的程序
#include"c1.h"
#include"func7-1.cpp" // 包括顶点信息类型的定义及对它的操作
#include"func7-2.cpp" // 包括弧(边)的相关信息类型的定义及对它的操作
#include"c7-1.h" // 图的数组(邻接矩阵)存储结构
```

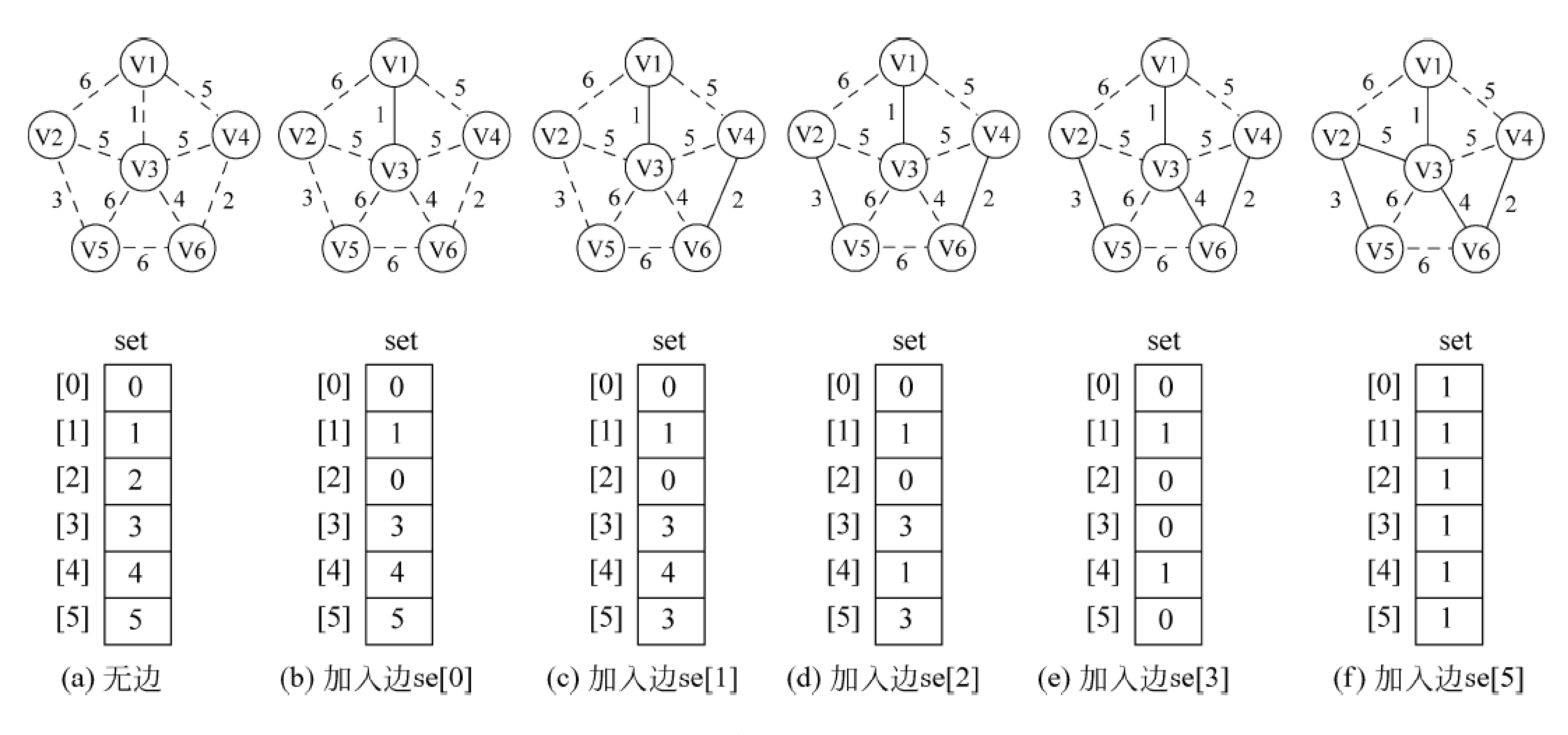
```
#include"bo7-1.cpp" // 图的数组(邻接矩阵)存储结构的基本操作
struct side //   图的边信息存储结构
{ int a,b; // 边的 2 顶点的序号
  VRType weight; // 边的权值
};
void Kruskal(MGraph G)
{ // 克鲁斯卡尔算法求无向连通网 G 的最小生成树
  int set[MAX_VERTEX_NUM],senumber = 0,sb,i,j,k;
  side se[MAX_VERTEX_NUM * (MAX_VERTEX_NUM - 1)/2]; // 存储边信息的一维数组
  for(i = 0;i<G.vexnum;++ i) // 查找所有的边,并根据权值升序插到 se 中
    for(j = i + 1;j<G.vexnum;++ j) // 无向网,只在上三角查找
      if(G.arcs[i][j].adj<INFINITY) // 顶点[i][j]之间有边
      { k = senumber - 1; // k 指向 se 的最后一条边
        while(k>= 0) //   k 仍指向 se 的边
          if(se[k].weight>G.arcs[i][j].adj)
          { // k 所指边的权值大于刚找到的边的权值
            se[k + 1] = se[k]; // k 所指的边向后移
            k --; // k 指向前一条边
          }
          else // k 所指边的权值不大于刚找到的边的权值
            break; // 跳出 while 循环
        se[k + 1].a = i; // 将刚找到的边的信息按权值升序插入 se
        se[k + 1].b = j;
        se[k + 1].weight = G.arcs[i][j].adj;
        senumber ++; // se 的边数 + 1
      }
  printf("i se[i].a se[i].b se[i].weight\n");
  for(i = 0;i<senumber;i ++ )
    printf("% d % 4d % 7d % 9d\n",i,se[i].a,se[i].b,se[i].weight);
  for(i = 0;i<G.vexnum;i ++ ) // 对于所有顶点
    set[i] = i; // 设置初态,各顶点分别属于各个集合
  printf("最小代价生成树的各条边为\n");
  j = 0; // j 指示 se 当前要并入最小生成树的边的序号,初值为 0
  k = 0; // k 指示当前构成最小生成树的边数
  while(k<G.vexnum - 1) // 最小生成树应有 G.vexnum - 1 条边
  { if(set[se[j].a]! = set[se[j].b]) // j 所指边的 2 顶点不属于同一个集合
    { printf("(% s - % s)\n",G.vexs[se[j].a].name,G.vexs[se[j].b].name); // 输出该边
      sb = set[se[j].b]; // 将该边的顶点 se[j].b 当前所在的集合赋给 sb
      for(i = 0;i<G.vexnum;i ++ ) // 对于所有顶点
        if(set[i] == sb) // 与顶点 se[j].b 在同一个集合中
          set[i] = set[se[j].a]; // 将此顶点并入顶点 se[j].a 所在的集合中
      k ++; // 当前构成最小生成树的边数 + 1
    }
    j ++; // j 指示 se 下一条要并入最小生成树的边的序号
```

```
    }
}
void main()
{
    MGraph g;
    char filename[13]; // 存储数据文件名(包括路径)
    printf("请输入数据文件名:");
    scanf("%s",filename);
    CreateFromFile(g,filename,0); // 创建无相关信息的网
    Display(g); // 输出无向网 g
    Kruskal(g); // 用克鲁斯卡尔算法输出 g 的最小生成树的各条边
}
```

**程序运行结果**(以教科书图 7.16 为例):

```
请输入数据文件名:f7-3.txt↙
6 个顶点 10 条边的无向网。顶点依次是:V1 V2 V3 V4 V5 V6 (见图 7-43)
G.arcs.adj:
      32767        6        1        5    32767    32767
          6    32767        5    32767        3    32767
          1        5    32767        5        6        4
          5    32767        5    32767    32767        2
      32767        3        6    32767    32767        6
      32767    32767        4        2        6    32767
G.arcs.info:
顶点 1  顶点 2  该边的信息:
i    se[i].a    se[i].b    se[i].weight
0      0          2            1
1      3          5            2
2      1          4            3
3      2          5            4
4      0          3            5
5      1          2            5
6      2          3            5
7      0          1            6
8      2          4            6
9      4          5            6
最小代价生成树的各条边为
(V1 - V3)
(V4 - V6)
(V2 - V5)
(V3 - V6)
(V2 - V3)
```

图 7-45 是以上程序运行的说明,显示了克鲁斯卡尔算法求最小生成树的过程。首先,主程序构造了图 7-43 所示的无向网。然后,调用 Kruskal(),求该网的最小生成树。其中用到了 2 个辅助数组 se[]和 set[]。se[]按照边的权值升序存储边的信息,具体内容见程序运行结果。set[i]表示第 i 个顶点所在的集合。设初态 set[i]=i,6 个顶点分属于 6 个集合,如图 7-45(a)所示。由 se[0](权值最小的边)开始,将边逐条加入到最小生成树中。加入边的原则有 2 个:




| | (a) 无边 | (b) 加入边se[0] | (c) 加入边se[1] | (d) 加入边se[2] | (e) 加入边se[3] | (f) 加入边se[5] |
|---|---|---|---|---|---|---|
| [0] | 0 | 0 | 0 | 0 | 0 | 1 |
| [1] | 1 | 1 | 1 | 1 | 1 | 1 |
| [2] | 2 | 0 | 0 | 0 | 0 | 1 |
| [3] | 3 | 3 | 3 | 3 | 0 | 1 |
| [4] | 4 | 4 | 4 | 1 | 1 | 1 |
| [5] | 5 | 5 | 3 | 3 | 0 | 1 |

图 7-45 运行 algo7-5.cpp 过程

(1) 边的权值尽量小,这一点是通过将 se[]按照边的权值升序存储,且由 se[0]开始逐边加入实现的;

(2) 所加入的边,其 2 顶点必须不在同一个集合中。

根据这个原则,首先将 se[0],即边(V1—V3)加入到最小生成树中,将 V1 和 V3 并到 1 个集合中。方法是将 V3 的集合 set[2]赋值为 set[0]( V1 的集合),并输出该边,如图 7-45(b)所示。用此方法依次将 se[1]、se[2]和 se[3] 三条边加入到最小生成树中,如图 7-45(c)、(d)和(e)所示。这时,考察 se[4],即边(V1—V4),虽然它是当前权值最小的边,但不满足原则(2),set[0]=set[3]=0,说明 se[4]的 2 顶点 V1 和 V4 在同一个集合中。舍弃 se[4],考察下一条边 se[5],即边(V2—V3)。se[5]满足加入边的原则,将 se[5]加入到最小生成树中,如图 7-45(f)所示。具有 6 个顶点的无向网,加入了 5 条满足原则(1)和原则(2)的边,构成了最小生成树。所构成的最小生成树和普里姆算法的一样。

## 7.3.3 关节点和重连通分量

```
// algo7-6.cpp 实现算法 7.10 和算法 7.11 的程序
#include"c1.h"
#include"func7-1.cpp" // 包括顶点信息类型的定义及对它的操作
#include"func7-4.cpp" // 弧(边)的相关信息类型的定义及对它的操作
#include"c7-2'.h" // 图的邻接表存储结构(与单链表的变量类型建立联系)
#include"bo7-2.cpp" // 图的邻接表存储结构的基本操作
```

```
int count,lowcount = 1; // 全局量 count 对访问顺序计数,lowcount 对求得 low 值的顺序计数
int low[MAX_VERTEX_NUM],lowOrder[MAX_VERTEX_NUM];
// 全局数组,low[]存顶点的 low 值,lowOrder 存顶点求得 low 值的顺序
int visited[MAX_VERTEX_NUM]; // 访问标志数组(全局量)
void DFSArticul(ALGraph G,int v0)
{ // 从第 v0 个顶点出发深度优先遍历图 G,查找并输出关节点。算法 7.11
  int min,w;
  ArcNode * p;
  visited[v0] = min = ++count; // v0 是第 count 个访问的顶点,min 的初值为 v0 的访问顺序
  for(p = G.vertices[v0].firstarc;p;p = p->nextarc) // 依次对 v0 的每个邻接顶点检查
  { w = p->data.adjvex; // w 为 v0 的邻接顶点位置
    if(visited[w] == 0) // w 未曾访问,是 v0 的孩子
    { DFSArticul(G,w);
      // 从第 w 个顶点出发深度优先遍历图 G,查找并输出关节点。返回前求得 low[w]
      if(low[w]<min) // 如果 v0 的孩子结点 w 的 low[]小,这说明孩子结点还与其他结点(祖先)相邻
        min = low[w]; // 取 min 值为孩子结点的 low[],则 v0 不是关节点
      else if(low[w]>= visited[v0]) // v0 的孩子结点 w 只与 v0 相连,则 v0 是关节点
        printf("%d %s\n",v0,G.vertices[v0].data.name); // 输出关节点 v0
    }
    else if(visited[w]<min) // w 已访问,则 w 是 v0 在生成树上的祖先,它的访问顺序必小于 min
      min = visited[w]; // 故取 min 为 visited[w]
  }
  low[v0] = min; // v0 的 low[]值为三者中的最小值
  lowOrder[v0] = lowcount++;
  // 记录 v0 求得 low[]值的顺序,总是在返回主调函数之前求得 low[]。新增
}
void FindArticul(ALGraph G)
{ // 连通图 G 以邻接表作存储结构,查找并输出 G 上全部关节点。全局量 count 对访问计数。算法 7.10
  int i,v;
  ArcNode * p;
  count = 1; // 访问顺序
  visited[0] = count; // 设定邻接表上 0 号顶点为生成树的根,第 1 个被访问
  for(i = 1;i<G.vexnum;++i) // 对于其余顶点
    visited[i] = 0; // 其余顶点尚未访问,设初值为 0
  p = G.vertices[0].firstarc; // p 指向根结点的第 1 个邻接顶点
  v = p->data.adjvex; // v 是根结点的第 1 个邻接顶点的序号
  DFSArticul(G,v); // 从第 v 顶点出发深度优先查找关节点
  if(count<G.vexnum) // 由根结点的第 1 个邻接顶点深度优先遍历 G,访问的顶点数少于 G 的顶点数
  { // 说明生成树的根有至少两棵子树,则根是关节点
    printf("%d %s\n",0,G.vertices[0].data.name); // 根是关节点,输出根
    while(p->nextarc) // 根有下一个邻接点
    { p = p->nextarc; // p 指向根的下一个邻接点
      v = p->data.adjvex;
      if(visited[v] == 0) // 此邻接点未被访问
```

```
            DFSArticul(G,v); // 从此顶点出发深度优先查找关节点
        }
    }
}
void main()
{
    int i;
    ALGraph g;
    char filename[13]; // 存储数据文件名(包括路径)
    printf("请输入数据文件名:");
    scanf("%s",filename);
    CreateFromFile(g,filename); // 由文件构造无向图 g
    Display(g); // 输出无向图 g
    printf("输出关节点:\n");
    FindArticul(g); // 求连通图 g 的关节点
    printf("i G.vertices[i].data visited[i] low[i] lowOrder[i]\n"); // 输出辅助变量
    for(i = 0;i<G.vexnum;++i)
        printf("%2d %9s %14d %8d %8d\n",i,g.vertices[i].data.name,
        visited[i],low[i],lowOrder[i]);
}
```

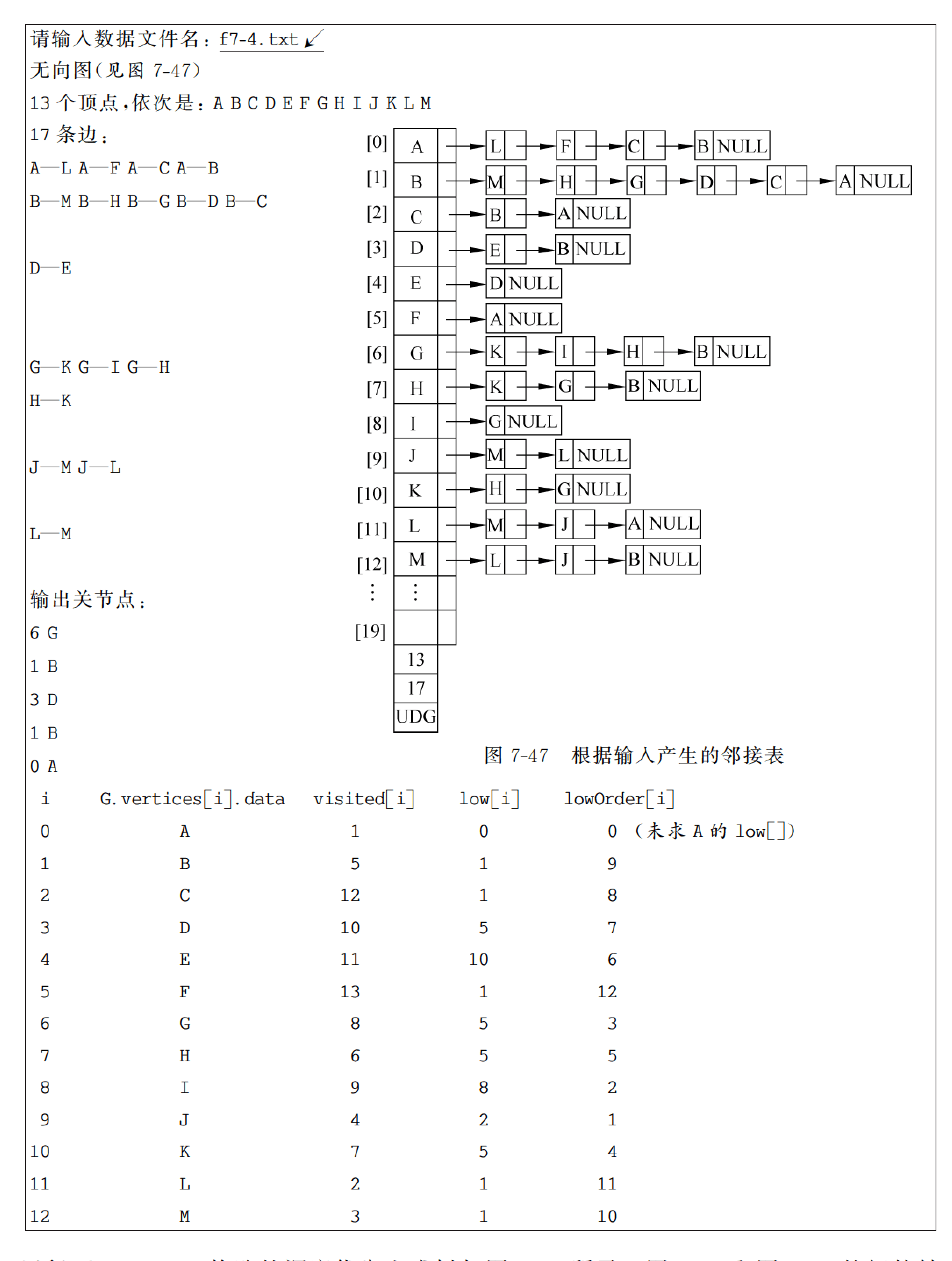
数据文件 f7-4.txt 的内容(图 7-46 是其所表示的无向图):

```
2
13
A B C D E F G H I J K L M
17
A B
A C
A F
A L
B C
B D
B G
B H
B M
D E
G H
G I
G K
H K
J L
J M
L M
```




图 7-46 数据文件 f7-4.txt 所表示的连通无向图

**程序运行结果**（以教科书中图 7.19 和图 7.20 为例）：



请输入数据文件名：f7-4.txt↙
无向图(见图 7-47)
13 个顶点，依次是：A B C D E F G H I J K L M
17 条边：
A—L A—F A—C A—B
B—M B—H B—G B—D B—C
D—E
G—K G—I G—H
H—K
J—M J—L
L—M
输出关节点：
6 G
1 B
3 D
1 B
0 A



图 7-47　根据输入产生的邻接表

| i | G.vertices[i].data | visited[i] | low[i] | lowOrder[i] |
|---|---|---|---|---|
| 0 | A | 1 | 0 | 0 (未求 A 的 low[]) |
| 1 | B | 5 | 1 | 9 |
| 2 | C | 12 | 1 | 8 |
| 3 | D | 10 | 5 | 7 |
| 4 | E | 11 | 10 | 6 |
| 5 | F | 13 | 1 | 12 |
| 6 | G | 8 | 5 | 3 |
| 7 | H | 6 | 5 | 5 |
| 8 | I | 9 | 8 | 2 |
| 9 | J | 4 | 2 | 1 |
| 10 | K | 7 | 5 | 4 |
| 11 | L | 2 | 1 | 11 |
| 12 | M | 3 | 1 | 10 |

运行 algo7-6.cpp 构造的深度优先生成树如图 7-48 所示。图 7-48 和图 7-46 的拓扑结构是一样的。5 条用虚线表示的边是构造深度优先生成树多余的边，它们是连接祖先的回边。如果一个结点不仅有连向双亲的边，还有连向祖先的回边。则对这个结点来说，它的双亲结点不是关节点。如图 7-48 中的结点 B，它有连向双亲结点 M 的边，还有连向祖先结点

A 的边。这样，如果删除结点 M，B 仍然与图的其他部分连通（重连通）。而图 7-48 中的结点 I，它只有连向双亲结点 G 的边。一旦结点 G 被删除，结点 I 就与图的其他部分不连通，也就是一个连通分量被分割成了多个连通分量。结点 G 被称为关节点。

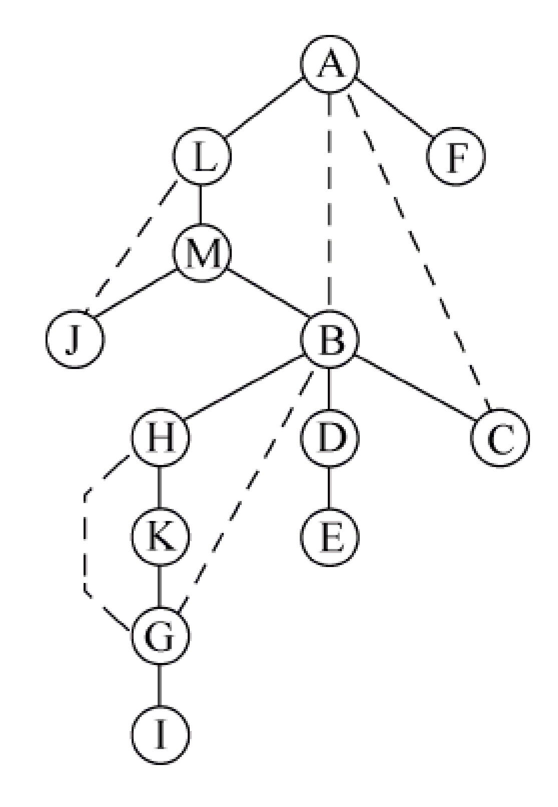


图 7-48　深度优先生成树

如何确定关节点？算法 7.10 和算法 7.11 的思路是这样的：首先在深度优先遍历图时，不仅标注某顶点是否被访问，还标注它的访问顺序。visited[]不再只是 FALSE 和 TRUE，而是1～顶点数。由于采用深度优先遍历，某结点的祖先被访问的顺序必先于该结点被访问的顺序。仍以图 7-48 为例，由第 1 个结点 A 深度优先遍历的顺序是：A、L、M、J、B、……，增加 1 个辅助数组 low[]，对顶点 v，定义 low[v]=min(visited[v]，low[w]，visited[k])。其中 w 和 k 分别是 v 的孩子和由回边相连的祖先。由算法 7.11 可知，low[]是在递归调用返回之前求得的。所以，求得 low[]的顺序是：……B、M、L、……，也就是说，孩子的 low[]是先于双亲的 low[]而获得的。这可由程序运行结果中的 lowOrder[]看出（增加辅助数组 lowOrder[]的目的就是帮助分析求得 low[]的顺序）。

如果顶点 v 有孩子 w，且有 low[w]≥visited[v]，则顶点 v 必为关节点。下面分析几种可能存在的情况：

(1) 如果顶点 v 有通过回边相连的祖先 k，则low[v]=visited[k]（祖先顶点 k 被访问的顺序）。同时 k 也是 v 的双亲 u 的祖先或双亲，故有 low[v]=visited[k]<visited[u]（结点祖先或双亲必先于该结点被访问）。不满足判定关节点的公式，故 u 不是 v 的关节点。这种情况如图 7-48 中顶点 B 的 low[]=顶点 A 的 visited[]=1，其双亲 M 的 visited[]=3，故 M 不是 B 的关节点。

(2) 如果顶点 v 没有通过回边相连的祖先，但有孩子 w，而孩子顶点 w 有通过回边相连的祖先 k，则 low[w]=visited[k]，而 k 也是 v 的双亲 u 的祖先，仍有 visited[k]≤visited[u]。如顶点 K 没有通过回边相连的祖先，但有孩子 G，而 G 有通过回边相连的祖先 B。顶点 G 的 low[]等于顶点 B 的 visited[]=5，也等于顶点 K 的 low[]。而 K 的双亲 H 的 visited[]=6，故 H 不是 K 的关节点。

(3) 如果顶点 v 既无孩子又无通过回边相连的祖先，则其双亲结点 u 是关节点。在这种情况下，low[v]=visited[v]（顶点 v 被访问的顺序）。而 u 被访问的顺序必定小于 v 的，故有 low[v]=visited[v]>visited[u]。所以 u 是 v 的关节点。如顶点 E 就是既无孩子又无通过回边相连的祖先，则其双亲结点 D 是 E 的关节点。

(4) 如果顶点 v 没有通过回边相连的祖先，虽有孩子顶点 w，但 w 也没有通过回边相连的祖先，则 v 的双亲结点 u 是关节点。在这种情况下，low[w]= visited[w]（顶点 w 被访问的顺序）>low[v]=visited[v]（顶点 v 被访问的顺序）>visited[u]，故 u 是 v 的关节点。如顶点 D 虽有孩子顶点 E，但 E 没有通过回边相连的祖先，则 low[D]=5=visited[B]。故 B 是 D 的关节点。

通过 low[w]≥visited[v]来判断连通图关节点的方法不能用于根结点。因为根结点的 visited[]=1，是最小值。判断根结点是否为关节点要看它有几棵子树，如果超过 1 棵，则根

结点就是关节点。原因是,它的每棵子树上的结点都和其他子树的结点不相连。否则在深度优先遍历其他子树时,就会遍历到,也就不成为根结点的子树了。所以算法 7.10 在深度优先遍历时,不是直接从根结点遍历,而是从根结点的第 1 个邻接顶点开始遍历。当遍历完这个邻接顶点的生成子树,若还有顶点未被访问,则说明根结点是关节点。如图 7-48 所示,对根结点 A 的第 1 棵子树 L 遍历结束后,A 还有邻接点 F 未被访问到。说明除根结点 A 之外,L 子树上的任何一个结点都不和 F 邻接。这样,若根结点 A 被删除,原图就会被分割成 L 子树和 F 两部分。故根结点 A 是关节点。

运行 algo7-6.cpp 在输出关节点时,B 被输出了 2 次。其原因是删除 B 使连通图分割成 3 个连通分量。

# 7.4 有向无环图及其应用

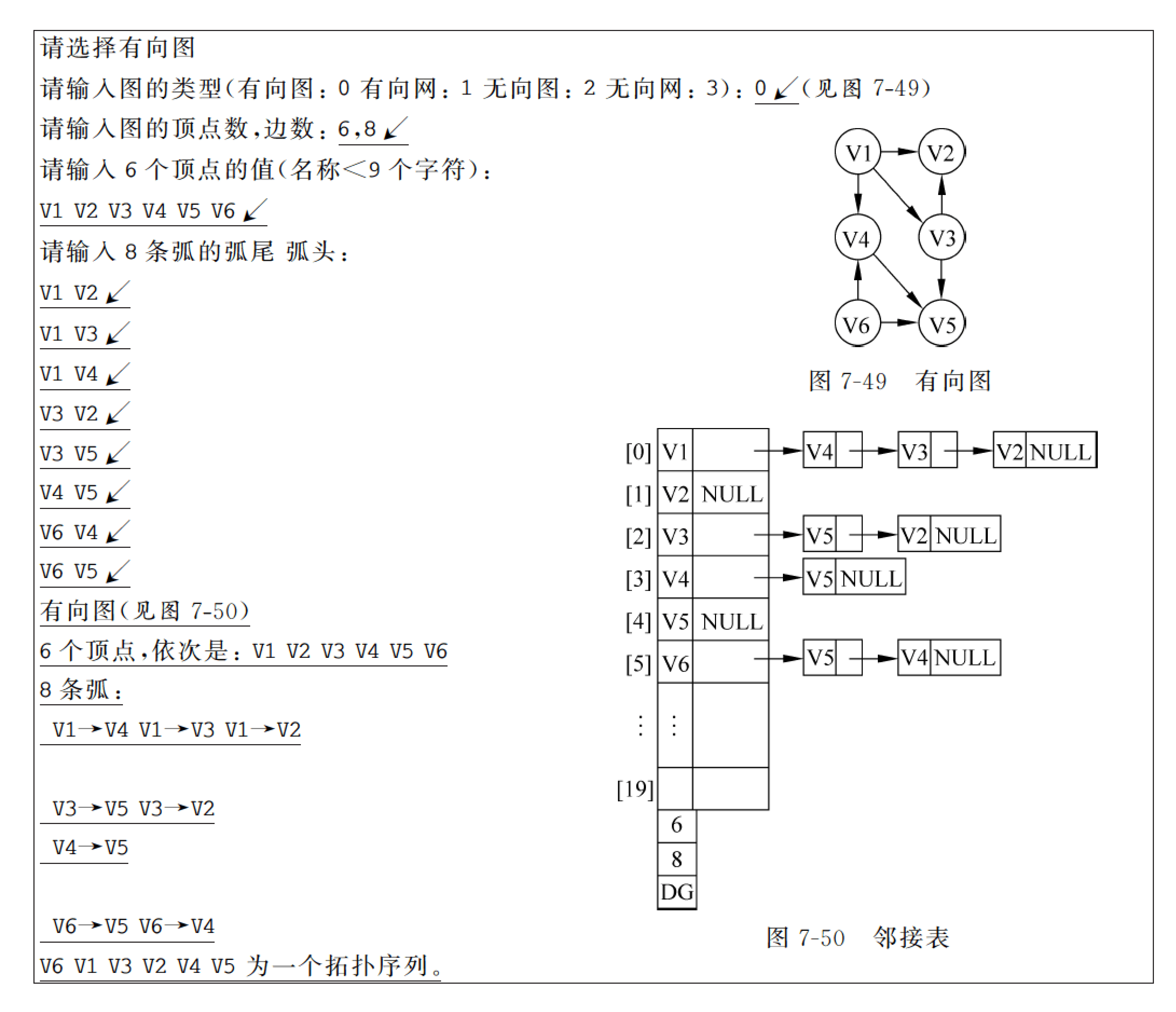
## 7.4.1 拓扑排序

```
// func7-5.cpp algo7-7.cpp 和 algo7-8.cpp 要调用
void FindInDegree(ALGraph G,int indegree[])
{ // 求顶点的入度,算法 7.12 和算法 7.13 调用
  int i;
  ArcNode *p;
  for(i=0;i<G.vexnum;i++) // 对于所有顶点
    indegree[i]=0; // 给顶点的入度赋初值 0
  for(i=0;i<G.vexnum;i++) // 对于所有顶点
  { p=G.vertices[i].firstarc; // p 指向顶点的邻接表的头指针
    while(p) // p 不空
    { indegree[p->data.adjvex]++; // 将 p 所指邻接顶点的入度 +1
      p=p->nextarc; // p 指向下一个邻接顶点
    }
  }
}
```

```
// algo7-7.cpp 输出有向图的一个拓扑序列。实现算法 7.12 的程序
#include"c1.h"
#include"func7-1.cpp" // 包括顶点信息类型的定义及对它的操作
#include"func7-4.cpp" // 弧(边)的相关信息类型的定义及对它的操作
#include"c7-2'.h" // 图的邻接表存储结构(与单链表的变量类型建立联系)
#include"bo7-2.cpp" // 图的邻接表存储结构的基本操作
#include"func7-5.cpp" // 求顶点入度的函数
typedef int SElemType; // 定义栈元素类型为整型(存储顶点序号)
#include"c3-1.h" // 顺序栈的存储结构
```

```
#include"bo3-1.cpp" // 顺序栈的基本操作
Status TopologicalSort(ALGraph G)
{ // 有向图 G 采用邻接表存储结构。若 G 无回路,则输出 G 的顶点的一个拓扑序列并返回 OK;
  // 否则返回 ERROR。算法 7.12
  int i,k,count = 0; // 已输出顶点数,初值为 0
  int indegree[MAX_VERTEX_NUM]; // 入度数组,存放各顶点当前入度数
  SqStack S;
  ArcNode * p;
  FindInDegree(G,indegree); // 对 G 的各顶点求入度 indegree[],在 func7-5.cpp 中
  InitStack(S); // 初始化零入度顶点栈 S
  for(i = 0;i<G.vexnum;++i) // 对所有顶点 i
    if(!indegree[i]) // 若其入度为 0
      Push(S,i); // 将 i 入零入度顶点栈 S
  while(!StackEmpty(S)) // 当零入度顶点栈 S 不空
  { Pop(S,i); // 出栈 1 个零入度顶点的序号,并将其赋给 i
    printf("%s",G.vertices[i].data.name); // 输出 i 号顶点
    ++ count; // 已输出顶点数 + 1
    for(p = G.vertices[i].firstarc;p;p = p->nextarc) // 对 i 号顶点的每个邻接顶点
    { k = p->data.adjvex; // 其序号为 k
      if(!(--indegree[k])) // k 的入度减 1,若减为 0,则将 k 入栈 S
        Push(S,k);
    }
  }
  if(count<G.vexnum) // 零入度顶点栈 S 已空,图 G 还有顶点未输出
  { printf("此有向图有回路\n");
    return ERROR;
  }
  else
  { printf("为一个拓扑序列。\n");
    return OK;
  }
}
void main()
{
  ALGraph f;
  printf("请选择有向图\n");
  CreateGraph(f); // 构造有向图 f,在 bo7-2.cpp 中
  Display(f); // 输出有向图 f,在 bo7-2.cpp 中
  TopologicalSort(f); // 输出有向图 f 的 1 个拓扑序列
}
```

**程序运行结果**(以教科书图 7.28 为例)：



```
请选择有向图
请输入图的类型(有向图：0 有向网：1 无向图：2 无向网：3)：0↙(见图 7-49)
请输入图的顶点数,边数：6,8↙
请输入 6 个顶点的值(名称<9 个字符)：
V1 V2 V3 V4 V5 V6↙
请输入 8 条弧的弧尾 弧头：
V1 V2↙
V1 V3↙
V1 V4↙
V3 V2↙
V3 V5↙
V4 V5↙
V6 V4↙
V6 V5↙
有向图(见图 7-50)
6 个顶点,依次是：V1 V2 V3 V4 V5 V6
8 条弧：
 V1→V4 V1→V3 V1→V2

 V3→V5 V3→V2
 V4→V5

 V6→V5 V6→V4
V6 V1 V3 V2 V4 V5 为一个拓扑序列。
```



图 7-49　有向图

图 7-50　邻接表

图 7-51 显示了运行 algo7-7.cpp 的过程。其中的 S 栈也可用队列代替,这样将输出一个不同的拓扑序列。

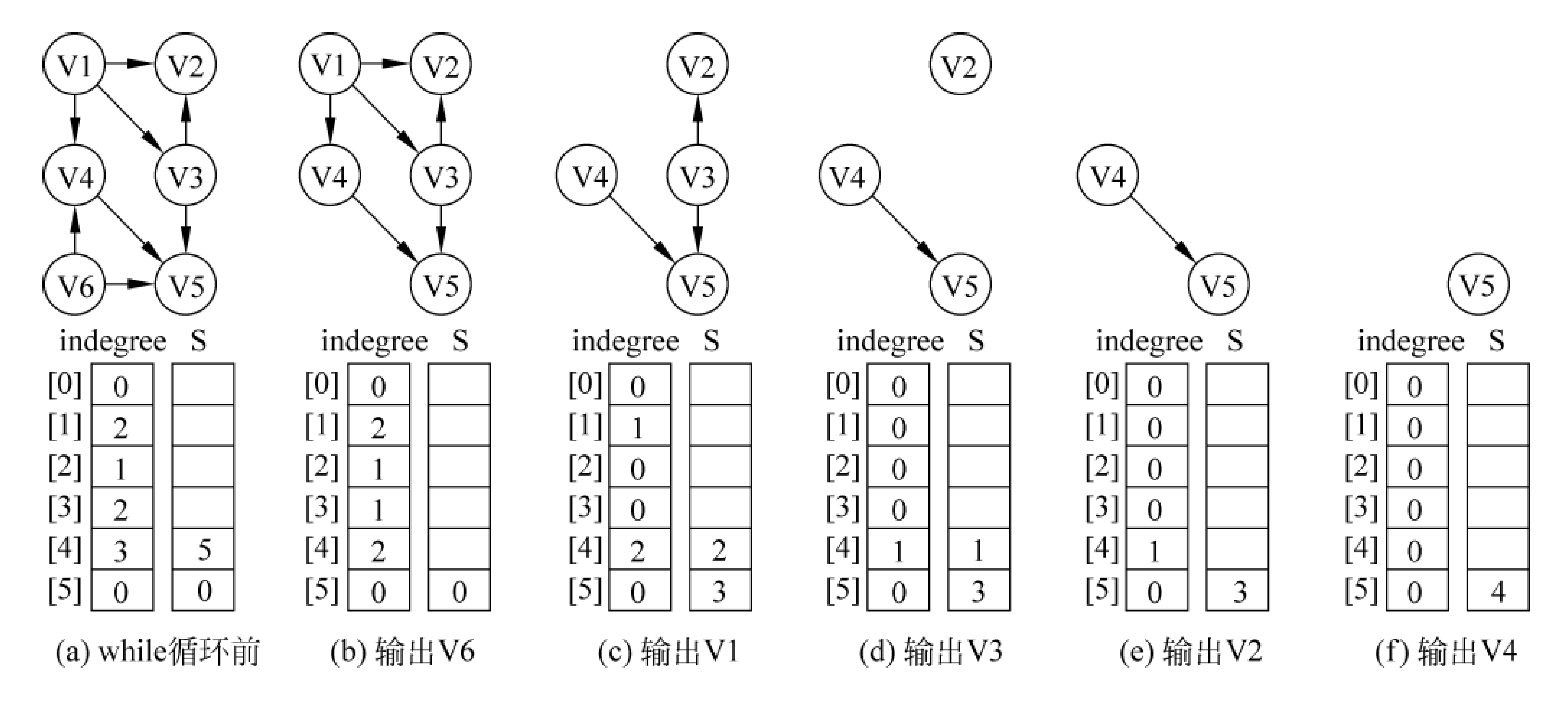



图 7-51　运行 algo7-7.cpp 过程

### 7.4.2 关键路径

算法 7.13 用到 2 个辅助数组：事件(顶点)最早发生时间 ve[]和事件最迟发生时间 vl[]。在 algo7-8.cpp 中，将这 2 个独立的、与顶点有关的数组绑定到顶点信息中，如 func7-6.cpp 所示。相比 func7-1.cpp，func7-6.cpp 的顶点信息类型多了 2 个成员 ve、vl。它们起到了辅助数组 ve[]和 vl[]的作用。func7-6.cpp 还多了一个对 ve、vl 操作的函数 Visitel()。

算法 7.14 用到了 ee、el 两个辅助变量。ee 表示活动(弧)的最早开始时间，el 表示活动的最迟开始时间(在不影响工期的情况下)。在 algo7-8.cpp 中，将这 2 个独立的、与弧有关的变量绑定到弧信息中。如 func7-7.cpp 所示(当然由于 ee、el 不是数组，这样做并不是必须的)。相比 func7-4.cpp，func7-7.cpp 的弧信息类型多了 2 个成员 ee、el，还多了一个对 ee、el 操作的函数 OutputArcwel()。

为了清楚地表示一个图，就要表明图的各顶点的所有信息、每 2 顶点之间的邻接关系(弧)的所有信息。将顶点信息、弧的信息及对顶点、弧的操作从图的基本操作中分离出来，成为一个独立的部分，就能使基本操作适用于图的顶点、弧的各种具体结构，从而提高了基本操作函数的利用率。

```
// func7-6.cpp 包括顶点信息类型的定义及对它的操作
#define MAX_NAME 9 // 顶点字符串的最大长度+1
struct VertexType // 顶点信息类型
{ char name[MAX_NAME]; // 顶点名称
  int ve,vl; // (顶点)事件最早发生时间,事件最迟发生时间
};
void Visit(VertexType ver) // 访问顶点名称的函数,输出图要用到
{ printf("%s",ver.name);
}
void Input(VertexType &ver) // 输入顶点信息的函数,创建图要用到
{ scanf("%s",ver.name);
}
void Visitel(VertexType ver) // 输出顶点 ve、vl 域的函数
{ printf("%3d%3d",ver.ve,ver.vl);
}
void InputFromFile(FILE* f,VertexType &ver) // 从文件输入顶点信息的函数
{ fscanf(f,"%s",ver.name);
}
```

```
// func7-7.cpp 包括弧的相关信息类型的定义及对它的操作
typedef int VRType; // 定义权值类型为整型
struct InfoType // 弧的相关信息类型
{ VRType weight; // 权值
  int ee,el; // (活动)最早开始时间,最迟开始时间
};
void InputArc(InfoType* &arc) // 动态生成弧的相关信息的空间并输入权值的函数,创建图用到
```

```
{ arc=(InfoType*)malloc(sizeof(InfoType)); // 动态生成存放弧信息的空间
  scanf("%d",&arc->weight); // 输入权值
}
void OutputArc(InfoType*arc) // 输出弧的权值的函数,输出图要用到
{ printf(":%d",arc->weight);
}
void OutputArcwel(InfoType*arc) // 输出弧的权值、ee 和 el 的函数
{ printf("%3d %3d %3d ",arc->weight,arc->ee,arc->el); // 输出弧的权值、ee 和 el
}
void InputArcFromFile(FILE*f,InfoType*&arc) // 由文件输入弧(边)的相关信息的函数
{ arc=(InfoType*)malloc(sizeof(InfoType)); // 动态生成存放弧(边)信息的空间
  fscanf(f,"%d",&arc->weight);
}
```

```
// algo7-8.cpp 求关键路径。实现算法 7.13、算法 7.14 的程序
#include"c1.h"
#include"func7-6.cpp" // 包括顶点信息类型的定义及对它的操作
#include"func7-7.cpp" // 弧的相关信息类型的定义及对它的操作
#include"c7-2'.h" // 图的邻接表存储结构(与单链表的变量类型建立联系)
#include"bo7-2.cpp" // 图的邻接表存储结构的基本操作
#include"func7-5.cpp" // 求顶点入度的函数
typedef int SElemType; // 定义栈元素类型为整型(存储顶点序号)
#include"c3-1.h" // 顺序栈的存储结构
#include"bo3-1.cpp" // 顺序栈的基本操作
Status TopologicalOrder(ALGraph &G,SqStack &T)
{ // 有向网 G 采用邻接表存储结构,求各顶点事件的最早发生时间 ve(存储在 G 中)。修改算法 7.13
  // T 为拓扑序列顶点栈,S 为零入度顶点栈。若 G 无回路,则用栈 T 返回 G 的一个拓扑序列,
  // 且函数值为 OK;否则为 ERROR
  int i,k,count=0; // 已入栈顶点数,初值为 0
  int indegree[MAX_VERTEX_NUM]; // 入度数组,存放各顶点当前入度数
  SqStack S;
  ArcNode *p;
  FindInDegree(G,indegree); // 对各顶点求入度 indegree[],在 func7-5.cpp 中
  InitStack(S); // 初始化零入度顶点栈 S
  printf("拓扑序列:");
  for(i=0;i<G.vexnum;++i) // 对所有顶点 i
    if(!indegree[i]) // 若其入度为 0
      Push(S,i); // 将 i 入零入度顶点栈 S
  InitStack(T); // 初始化拓扑序列顶点栈
  for(i=0;i<G.vexnum;++i) // 初始化 ve=0(最小值,先假定每个事件都不受其他事件约束)
    G.vertices[i].data.ve=0;
  while(!StackEmpty(S)) // 当零入度顶点栈 S 不空
  { Pop(S,i); // 从栈 S 将已拓扑排序的顶点弹出,并赋给 i
    Visit(G.vertices[i].data); // 输出该顶点的名称
    Push(T,i); // j 号顶点入逆拓扑排序栈 T(栈底元素为拓扑排序的第 1 个元素)
```

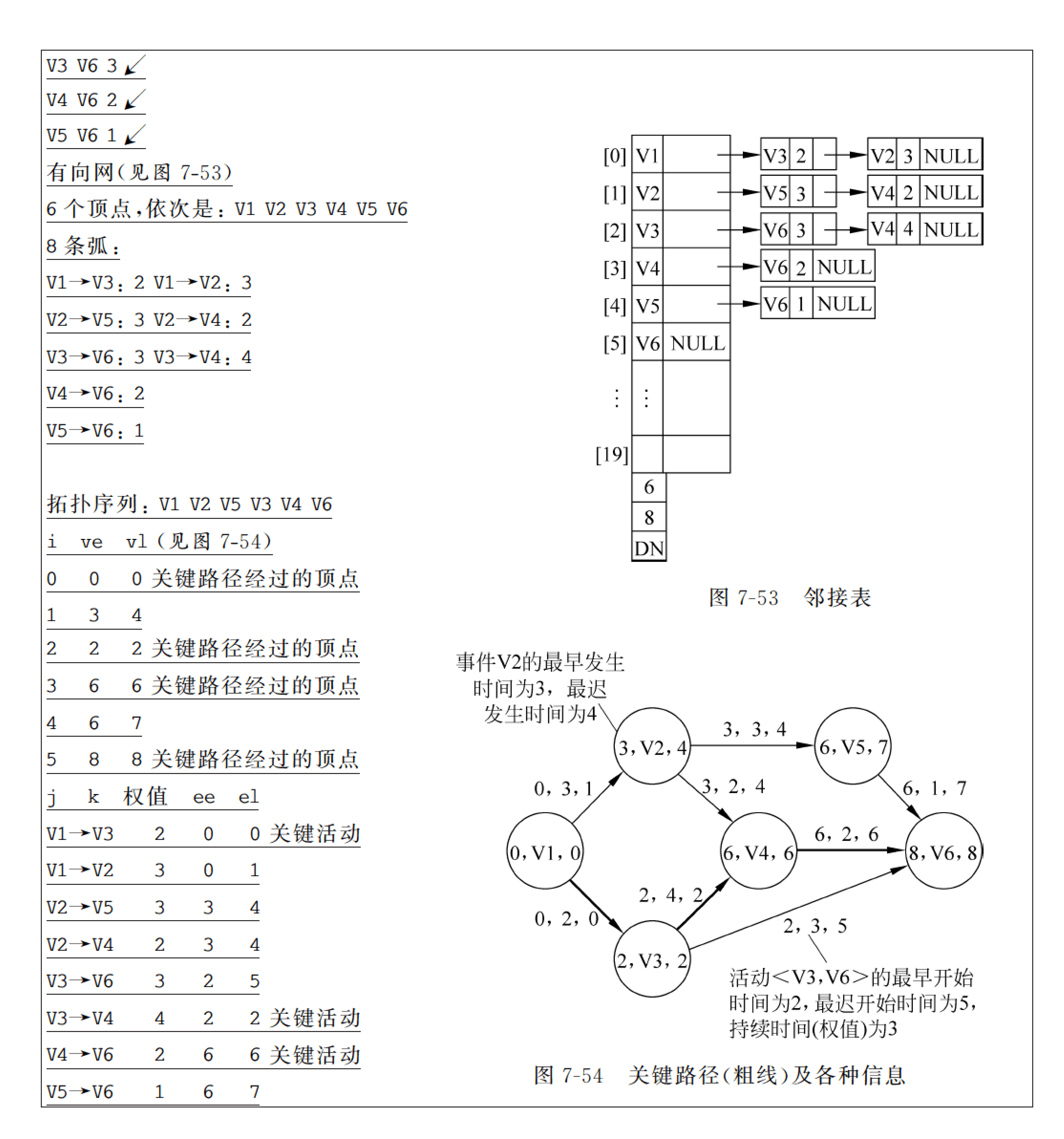
```
      ++count; // 对入栈 T 的顶点计数
      for(p = G.vertices[i].firstarc;p;p = p->nextarc)
      { // 对 i 号顶点的每个邻接顶点
        k = p->data.adjvex; // 其序号为 k
        if( --indegree[k] == 0) // k 的入度减 1,若减为 0,则将 k 入栈 S
          Push(S,k);
        if(G.vertices[i].data.ve + p->data.info->weight>G.vertices[k].data.ve)
        // 顶点 i 事件的最早发生时间 + <i,k>的权值>顶点 k 事件的最早发生时间
           G.vertices[k].data.ve = G.vertices[i].data.ve + p->data.info->weight;
           // 顶点 k 事件的最早发生时间 = 顶点 i 事件的最早发生时间 + <i,k>的权值
      }    // 由于 i 已拓扑有序,故 G.vertices[i].data.ve 不再改变
    }
    if(count<G.vexnum)
    { printf("此有向网有回路\n");
      return ERROR;
    }
    else
      return OK;
  }
  Status CriticalPath(ALGraph &G)
  { // G 为有向网,输出 G 的各项关键活动。修改算法 7.14
    SqStack T;
    int i,j,k;
    ArcNode * p;
    if(!TopologicalOrder(G,T)) // 产生有向环
      return ERROR;
    j = G.vertices[0].data.ve; // j 的初值
    for(i = 1;i<G.vexnum;i++ ) // 在所有顶点中,找 ve 的最大值
      if(G.vertices[i].data.ve>j)
        j = G.vertices[i].data.ve; // j = Max(ve) 完成点的最早发生时间
    for(i = 0;i<G.vexnum;i++ ) // 初始化顶点事件的最迟发生时间
      G.vertices[i].data.vl = j; // 为完成点的最早发生时间(最大值)
    while(!StackEmpty(T)) // 按拓扑逆序求各顶点的 vl 值
      for(Pop(T,j),p = G.vertices[j].firstarc;p;p = p->nextarc)
      { // 弹出栈 T 的元素,赋给 j,p 指向顶点 j 的后继事件(出弧)顶点 k,
        // 事件 k 的最迟发生时间已确定(因为是逆拓扑排序)
        k = p->data.adjvex; // 后继事件顶点的序号
        if(G.vertices[k].data.vl - p->data.info->weight<G.vertices[j].data.vl)
        // 事件 j 的最迟发生时间>其直接后继事件 k 的最迟发生时间 - <j,k>的权值
           G.vertices[j].data.vl = G.vertices[k].data.vl - p->data.info->weight;
           // 事件 j 的最迟发生时间 = 事件 k 的最迟发生时间 - <j,k>的权值
      }    // 由于 k 已逆拓扑有序,故 G.vertices[k].data.vl 不再改变
    printf("\ni  ve  vl\n");
    for(i = 0;i<G.vexnum;i++ ) // 对于每个顶点
    { printf("%d",i); // 输出序号
```

```
    Visitel(G.vertices[i].data); // 输出 ve、vl 值,在 func7-6.cpp 中
    if(G.vertices[i].data.ve == G.vertices[i].data.vl)
    // 事件(顶点)的最早发生时间 = 最迟发生时间
      printf(" 关键路径经过的顶点");
    printf("\n");
  }
  printf("j   k   权值   ee   el\n"); // 以下求 ee,el 和关键活动
  for(j = 0;j<G.vexnum;++j) // 对于每个顶点 j
    for(p = G.vertices[j].firstarc;p;p = p->nextarc)
    { // p 依次指向其邻接顶点(直接后继事件)
      k = p->data.adjvex; // 邻接顶点(直接后继事件)序号
      p->data.info->ee = G.vertices[j].data.ve;
      // ee(活动<j,k>的最早开始时间) = (顶点 j)事件最早发生时间
      p->data.info->el = G.vertices[k].data.vl - p->data.info->weight;
      // el(活动<j,k>的最迟开始时间) = (顶点 k)事件最迟发生时间 - <j,k>的权值
      printf("%s→%s",G.vertices[j].data.name,G.vertices[k].data.name); // 输出弧
      OutputArcwel(p->data.info); // 输出弧的权值、ee 和 el,在 func7-7.cpp 中
      if(p->data.info->ee == p->data.info->el)
      // 活动(弧)的最早开始时间 = 活动的最迟开始时间
        printf("关键活动");
      printf("\n");
    }
  return OK;
}
void main()
{
  ALGraph h;
  printf("请选择有向网\n");
  CreateGraph(h); // 构造有向网 h,在 bo7-2.cpp 中
  Display(h); // 输出有向网 h,在 bo7-2.cpp 中
  CriticalPath(h); // 求 h 的关键路径
}
```

**程序运行结果**(以教科书中图 7.30 为例):

```
请选择有向网
请输入图的类型(有向图:0 有向网:1 无向图:2 无向网:3):1↙(见图 7-52)
请输入图的顶点数,边数:6,8↙
请输入 6 个顶点的值(名称<9 个字符):
V1 V2 V3 V4 V5 V6↙
请输入 8 条弧的弧尾 弧头 弧的信息:
V1 V2 3↙
V1 V3 2↙
V2 V4 2↙
V2 V5 3↙
V3 V4 4↙
```


图 7-52　有向网



```
V3 V6 3↙
V4 V6 2↙
V5 V6 1↙
有向网(见图 7-53)
6 个顶点,依次是: V1 V2 V3 V4 V5 V6
8 条弧:
V1→V3: 2 V1→V2: 3
V2→V5: 3 V2→V4: 2
V3→V6: 3 V3→V4: 4
V4→V6: 2
V5→V6: 1

拓扑序列: V1 V2 V5 V3 V4 V6
i  ve  vl (见图 7-54)
0  0   0 关键路径经过的顶点
1  3   4
2  2   2 关键路径经过的顶点
3  6   6 关键路径经过的顶点
4  6   7
5  8   8 关键路径经过的顶点
j      k  权值  ee  el
V1→V3     2     0   0 关键活动
V1→V2     3     0   1
V2→V5     3     3   4
V2→V4     2     3   4
V3→V6     3     2   5
V3→V4     4     2   2 关键活动
V4→V6     2     6   6 关键活动
V5→V6     1     6   7
```



图 7-53 邻接表



图 7-54 关键路径(粗线)及各种信息

图 7-53 是由输入产生的邻接表。为直观和方便起见,表结点中邻接顶点的序号直接用其名称代替,动态生成的权值也直接写在相关信息的指针域中。

顶点 i 的事件最早发生时间 ve[i]取决于其直接前驱事件的发生时间和二者之间活动的持续时间(弧的权值)。以图 7-52 为例,V2 事件的最早发生时间 ve[]取决于 V1 的 ve[]和＜V1,V2＞的权值 3。即 V2 的 ve[]等于 V1 的 ve[]+3。如果顶点 i 有多个直接前驱事件则 ve[i] 取它们的最大值,如 V4 的 ve[]取决于 V2 的 ve[]+＜V2,V4＞的权值 2 与 V3 的 ve[]+＜V3,V4＞的权值 4 这 2 者中的大值。没有直接前驱事件的顶点(源点,工程开始点),其 ve[]=0(是最小值),如顶点 V1。由于求顶点的 ve[]要求其直接前驱的 ve[]已知,故应先对有向网进行拓扑排序。排序前设所有顶点的 ve[]初值=0(最小值),当出现较大的值,则用这个大值更新 ve[]。调用 TopologicalOrder()后,ve[]如以上程序运行结果所示。

所谓事件最迟发生时间是指在不影响工期的情况下,某事件可以最迟发生的时间。顶点 i 的事件最迟发生时间 vl[i]取决于其直接后继事件的最迟发生时间和二者之间活动的持

续时间(弧的权值)。仍以图 7-52 为例,V4 事件的最迟发生时间 vl[]取决于 V6 的 vl[]和＜V4,V6＞的权值 2。即 V4 的 vl[]等于 V6 的 vl[]－2。如果顶点 i 有多个直接后继事件则 vl[i]取最小值。如 V3 的 vl[]取决于 V4 的 vl[]－＜V3,V4＞的权值 4 与 V6 的 vl[]－＜V3,V6＞的权值3 这 2 者中的小值。没有直接后继事件的顶点(汇点,工程完成点),其 vl[]＝ve[](是最大值,已先期求出),如顶点 V6。由于求顶点的 vl[]时,要求其直接后继的 vl[]已知,故应先形成有向网的逆拓扑序列。TopologicalOrder()将已拓扑排序的顶点入栈 T,形成逆拓扑序列。排序前设所有顶点的 vl[]初值等于没有后继的那个顶点的 vl[](最大值),本例中这个顶点是 V6。当出现较小的值,则用这个小值更新 vl[]。调用 CriticalPath(),vl[]如以上程序运行结果所示。

若对于顶点 i,有 ve[i]＝vl[i],即事件最早发生时间等于事件最迟发生时间。说明为保证工期,事件(顶点)i 的发生时间不可变更。如果变小,则前面的活动(入弧)还未完成;如果变大,则影响后继事件按时完成。因此,顶点 i 是关键路径要经过的点。如以上程序运行结果所示,V1、V3、V4 和 V6 是关键路径要经过的顶点。但仅根据这些顶点,还不能确定关键路径。如图 7-52 中,虽然 V3、V4 和 V6 是关键路径要经过的顶点,但弧＜V3,V4＞、＜V3,V6＞和＜V4,V6＞中哪个是关键路径还不清楚。

对于一个活动(弧)＜j,k＞,它的最早开始时间 ee 等于它的前端事件(顶点 j)的最早发生时间 ve[j];它的最迟开始时间 el 等于它的后端事件(顶点 k)的最迟发生时间 vl[k]减去＜j,k＞的权值;如果活动＜j,k＞的 ee＝el,那么这个活动的开始时间就没有变动的余地,它就是整个关键路径的一部分。求得 ve[]和 vl[]后,对于每一个弧＜j,k＞,判断它的 ee 是否等于 el,可求出所有关键路径。图 7-54 中用粗箭头表示关键路径。

# 7.5　最短路径

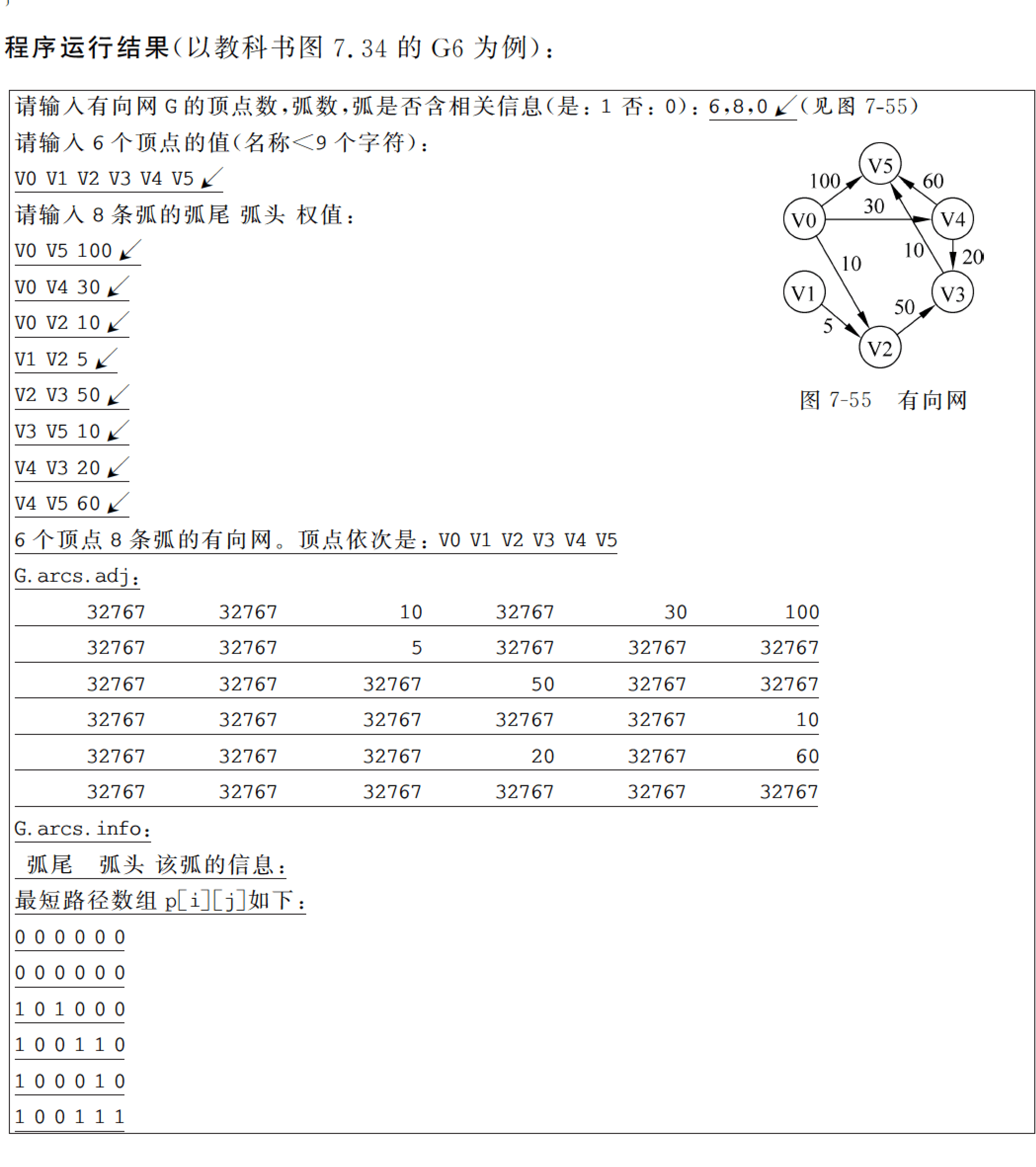
## 7.5.1　从某个源点到其余各顶点的最短路径

```
// algo7-9.cpp 实现算法 7.15 的程序。迪杰斯特拉算法的实现
#include"c1.h"
#include"func7-1.cpp" // 包括顶点信息类型的定义及对它的操作
#include"func7-2.cpp" // 包括弧(边)的相关信息类型的定义及对它的操作
#include"c7-1.h" // 图的数组(邻接矩阵)存储结构
#include"bo7-1.cpp" // 图的数组(邻接矩阵)存储结构的基本操作
typedef Status PathMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM]; // 路径矩阵,二维数组
typedef VRType ShortPathTable[MAX_VERTEX_NUM]; // 最短距离表,一维数组
void ShortestPath_DIJ(MGraph G,int v0,PathMatrix P,ShortPathTable D)
{ // 用 Dijkstra 算法求有向网 G 的 v0 顶点到其余顶点 v 的最短路径 P[v]及带权长度 D[v]。
  // 若 P[v][w]为 TRUE,则 w 是从 v0 到 v 当前求得最短路径上的顶点。
  // final[v]为 TRUE 当且仅当 v∈S,即已经求得从 v0 到 v 的最短路径。算法 7.15
  int v,w,i,j;
  VRType min;
```

```
    Status final[MAX_VERTEX_NUM];
    // 辅助矩阵,为真表示该顶点到 v0 的最短距离已求出,初值为假
    for(v = 0;v<G.vexnum;++v)
    { final[v] = FALSE; // 设初值
      D[v] = G.arcs[v0][v].adj; // D[]存放 v0 到 v 的最短距离,初值为 v0 到 v 的直接距离
      for(w = 0;w<G.vexnum;++w)
        P[v][w] = FALSE; // 设 P[][]初值为 FALSE,没有路径
      if(D[v]<INFINITY) // v0 到 v 有直接路径
        P[v][v0] = P[v][v] = TRUE;
        // 一维数组 p[v][]表示源点 v0 到 v 最短路径通过的顶点,目前通过 v0 和 v 两顶点
    }
    D[v0] = 0; // v0 到 v0 距离为 0
    final[v0] = TRUE; // v0 顶点并入 S 集
    for(i = 1;i<G.vexnum;++i) // 对于其余 G.vexnum - 1 个顶点
    { // 开始主循环,每次求得 v0 到某个顶点 v 的最短路径,并将 v 并入 S 集
      min = INFINITY; // 当前所知离 v0 顶点的最近距离,设初值为∞
      for(w = 0;w<G.vexnum;++w) // 对所有顶点检查
        if(!final[w]&&D[w]<min) // 在 S 集之外的顶点(其 final[] = FALSE)中
        { // 找离 v0 最近的顶点 w,并将其赋给 v,距离赋给 min
          v = w; // 在 S 集之外的离 v0 最近的顶点序号
          min = D[w]; // 最近的距离
        }
      final[v] = TRUE; // 将 v 并入 S 集
      for(w = 0;w<G.vexnum;++w) // 根据新并入的顶点,更新不在 S 集的顶点到 v0 的距离和路径数组
        if(!final[w]&&min<INFINITY&&G.arcs[v][w].adj<INFINITY&&(min +
        G.arcs[v][w].adj<D[w]))
        { // w 不属于 S 集且 v0→v→w 的距离<目前 v0→w 的距离
          D[w] = min + G.arcs[v][w].adj; // 更新 D[w]
          for(j = 0;j<G.vexnum;++j)
          // 修改 P[w],v0 到 w 经过的顶点包括 v0 到 v 经过的顶点再加上顶点 w
            P[w][j] = P[v][j];
          P[w][w] = TRUE;
        }
    }
  }
  void main()
  {
    int i,j;
    MGraph g;
    PathMatrix p; // 二维数组,路径矩阵
    ShortPathTable d; // 一维数组,最短距离表
    CreateDN(g); // 构造有向网 g
```

```
  Display(g); // 输出有向网 g
  ShortestPath_DIJ(g,0,p,d);
  // 以 g 中序号为 0 的顶点为源点,求其到其余各顶点的最短距离。存于 d 中
  printf("最短路径数组 p[i][j]如下:\n");
  for(i = 0;i<G.vexnum;++i)
  { for(j = 0;j<g.vexnum;++j)
      printf("%2d",p[i][j]);
    printf("\n");
  }
  printf("%s 到各顶点的最短路径长度为\n",g.vexs[0].name);
  for(i = 0;i<G.vexnum;++i)
    if(i!= 0)
      printf("%s→%s:%d\n",g.vexs[0].name,g.vexs[i].name,d[i]);
}
```

**程序运行结果**(以教科书图 7.34 的 G6 为例):



请输入有向网 G 的顶点数,弧数,弧是否含相关信息(是:1 否:0):6,8,0↙(见图 7-55)
请输入 6 个顶点的值(名称<9 个字符):
V0 V1 V2 V3 V4 V5↙
请输入 8 条弧的弧尾 弧头 权值:
V0 V5 100↙
V0 V4 30↙
V0 V2 10↙
V1 V2 5↙
V2 V3 50↙
V3 V5 10↙
V4 V3 20↙
V4 V5 60↙
6 个顶点 8 条弧的有向网。顶点依次是:V0 V1 V2 V3 V4 V5
G.arcs.adj:

| | | | | | |
|---|---|---|---|---|---|
| 32767 | 32767 | 10 | 32767 | 30 | 100 |
| 32767 | 32767 | 5 | 32767 | 32767 | 32767 |
| 32767 | 32767 | 32767 | 50 | 32767 | 32767 |
| 32767 | 32767 | 32767 | 32767 | 32767 | 10 |
| 32767 | 32767 | 32767 | 20 | 32767 | 60 |
| 32767 | 32767 | 32767 | 32767 | 32767 | 32767 |

G.arcs.info:
 弧尾　弧头 该弧的信息:
最短路径数组 p[i][j]如下:
0 0 0 0 0 0
0 0 0 0 0 0
1 0 1 0 0 0
1 0 0 1 1 0
1 0 0 0 1 0
1 0 0 1 1 1



图 7-55　有向网

```
V0 到各顶点的最短路径长度为
V0→V1：32767
V0→V2：10
V0→V3：50
V0→V4：30
V0→V5：60
```

函数 ShortestPath_DIJ()利用两个辅助数组 final[]和 D[]求得给定点 v0 到图 G 中其余各顶点的最短距离。D[]存放当前 v0 到其余各顶点的最短距离，final[]的初值为 FALSE。final[]的值为 TRUE，表示 v0 到该顶点的最短距离已求出。

图 7-56 通过 final[]和 D[]的变化演示了求解过程。最初，final[]的初值中只有 final[v0]为真，最短距离顶点集 S 中只有顶点 v0(源点，此例中实参为 V0)。D[]的初值是邻接矩阵中 v0 行所对应的值。另令 D[v0]=0(v0 到自己的距离当然为 0)。v0 到某顶点 i 的最短距离可能是二者的直接距离 G. arcs[v0][i]. adj，也可能是由 v0 出发，经过其他顶点，最后到达顶点 i 的距离。如图 7-55 中 V0 到 V5 的最短距离不是它们的直接距离 100，而是由 V0 经过 V4、V3，最后到达 V5 的距离 60。

| | (a) final | (a) D | (b) final | (b) D | (c) final | (c) D | (d) final | (d) D | (e) final | (e) D |
|---|---|---|---|---|---|---|---|---|---|---|
| [0] | TRUE | 0 | TRUE | 0 | TRUE | 0 | TRUE | 0 | TRUE | 0 |
| [1] | FALSE | ∞ | FALSE | ∞ | FALSE | ∞ | FALSE | ∞ | FALSE | ∞ |
| [2] | FALSE | 10 | TRUE | 10 | TRUE | 10 | TRUE | 10 | TRUE | 10 |
| [3] | FALSE | ∞ | FALSE | 60 | FALSE | 50 | TRUE | 50 | TRUE | 50 |
| [4] | FALSE | 30 | FALSE | 30 | TRUE | 30 | TRUE | 30 | TRUE | 30 |
| [5] | FALSE | 100 | FALSE | 100 | FALSE | 90 | FALSE | 60 | TRUE | 60 |

(a) 初值　(b) V2并入S　(c) V4并入S　(d) V3并入S　(e) V5并入S

图 7-56　运行 algo7-9. cpp 过程

根据图 7-56(a)，在不属于 S 集的顶点中，V0 到 V2 的距离最短。可以断定，V0 到 V2 的距离 10 是最短距离。V0 通过其他顶点绕道到达 V2 的距离一定会比 10 大，故将 V2 并入 S 集中(final[2]=TRUE)。同时考察 S 集外的顶点中，有没有哪个顶点 i，使得 V0 先到 V2(距离为 10)，再由 V2 到达顶点 i 比直接从 V0 到达顶点 i 的距离要小？也就是满足 10+G. arcs[2][i]. adj<D[i]。如有，则改写 D[i]。V0 本无直接到达 V3 的路径，但有 V0→V2→V3 的路径，为 10+G. arcs[2][3]. adj=10+50=60，故改写 D[3]=60，如图 7-56(b)所示。用这样的方法，依次将 V4、V3 和 V5 并入 S。详见图 7-56(c)、(d)和(e)。

通过 final[]和 D[]可求得给定点 v0 到图 G 中其余各顶点的最短距离是多少，但却不知道其间通过哪些顶点。矩阵 P[][]有这些顶点的信息。以程序运行结果为例，一维数组 p[2][]中的 1 是 V0 到 V2 经过的顶点(只有 V0 和 V2 两个顶点)；p[3][]中的 1 是 V0 到 V3 经过的顶点(V0、V3 和 V4)。

### 7.5.2 每一对顶点之间的最短路径

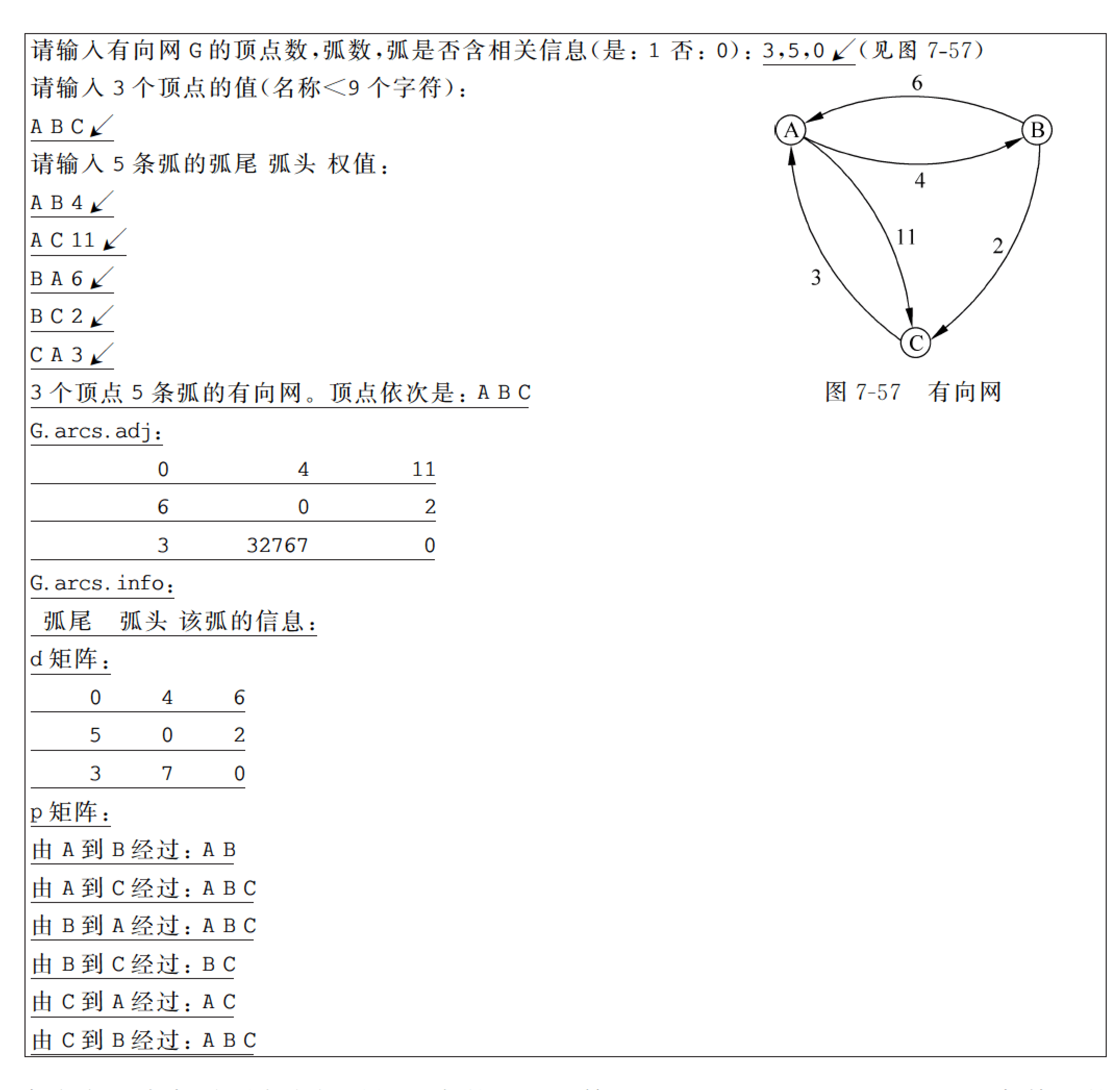
```
// func7-8.cpp 算法 7.16,algo7-10.cpp 和 algo7-11.cpp 用到
void ShortestPath_FLOYD(MGraph G,PathMatrix P,DistancMatrix D)
{ // 用 Floyd 算法求有向网 G 中各对顶点 v 和 w 之间的最短路径 P[v][w][]及其带权长度 D[v][w]。
```

```
  // 若 P[v][w][u]为 TRUE,则 u 是从 v 到 w 当前求得最短路径上的顶点。算法 7.16
  int u,v,w,i;
  for(v = 0;v<G.vexnum;v++ ) // 各对结点之间初始已知路径及距离
    for(w = 0;w<G.vexnum;w++ )
    { D[v][w] = G.arcs[v][w].adj; // 顶点 v 到顶点 w 的直接距离
      for(u = 0;u<G.vexnum;u++ )
        P[v][w][u] = FALSE; // 路径矩阵初值
      if(D[v][w]<INFINITY) // 从 v 到 w 有直接路径
        P[v][w][v] = P[v][w][w] = TRUE; // 由 v 到 w 的路径经过 v 和 w 两点
    }
  for(u = 0;u<G.vexnum;u++ )
    for(v = 0;v<G.vexnum;v++ )
      for(w = 0;w<G.vexnum;w++ )
        if(D[v][u]<INFINITY&&D[u][w]<INFINITY&&D[v][u] + D[u][w]<D[v][w])
        { // 从 v 经 u 到 w 的一条路径更短
          D[v][w] = D[v][u] + D[u][w]; // 更新最短距离
          for(i = 0;i<G.vexnum;i++ )
            P[v][w][i] = P[v][u][i]||P[u][w][i];
            // 从 v 到 w 的路径经过从 v 到 u 和从 u 到 w 的所有路径
        }
}
```

```
// algo7-10.cpp 实现算法 7.16 的程序
#include"c1.h"
#include"func7-1.cpp" // 包括顶点信息类型的定义及对它的操作
#include"func7-2.cpp" // 包括弧(边)的相关信息类型的定义及对它的操作
#include"c7-1.h" // 图的数组(邻接矩阵)存储结构
#include"bo7-1.cpp" // 图的数组(邻接矩阵)存储结构的基本操作
typedef char PathMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM][MAX_VERTEX_NUM];
// 三维数组,其值只可能是 0 或 1,故用 char 类型以减少存储空间的浪费
typedef VRType DistancMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM]; // 二维数组
#include"func7-8.cpp" // 求有向网中各对顶点之间最短距离的 Floyd 算法
void main()
{
  MGraph g;
  int i,j,k;
  PathMatrix p; // 三维数组
  DistancMatrix d; // 二维数组
  CreateDN(g); // 构造有向网 g
  for(i = 0;i<G.vexnum;i++ )
    g.arcs[i][i].adj = 0;
    // ShortestPath_FLOYD()要求对角元素值为 0,因为两点相同,其距离为 0
  Display(g); // 输出有向网 g
  ShortestPath_FLOYD(g,p,d); // 求每对顶点间的最短路径。在 func7-8.cpp 中
  printf("d 矩阵: \n");
  for(i = 0;i<G.vexnum;i++ )
  { for(j = 0;j<g.vexnum;j++ )
```

```
        printf("%6d",d[i][j]);
      printf("\n");
    }
    printf("p 矩阵:\n");
    for(i = 0;i<G.vexnum;i++ )
      for(j = 0;j<g.vexnum;j++ )
        if(i!= j)
        { printf("由 %s 到 %s 经过:",g.vexs[i].name,g.vexs[j].name);
          for(k = 0;k<g.vexnum;k++ )
            if(p[i][j][k] == 1)
              printf("%s",g.vexs[k].name);
          printf("\n");
        }
}
```

**程序运行结果**(以教科书中图 7.36 的 G7 为例):



请输入有向网 G 的顶点数,弧数,弧是否含相关信息(是:1 否:0):3,5,0↙(见图 7-57)
请输入 3 个顶点的值(名称<9 个字符):
A B C↙
请输入 5 条弧的弧尾 弧头 权值:
A B 4↙
A C 11↙
B A 6↙
B C 2↙
C A 3↙
3 个顶点 5 条弧的有向网。顶点依次是:A B C
G.arcs.adj:

| | | |
|---|---|---|
| 0 | 4 | 11 |
| 6 | 0 | 2 |
| 3 | 32767 | 0 |

G.arcs.info:
弧尾 弧头 该弧的信息:
d 矩阵:

| | | |
|---|---|---|
| 0 | 4 | 6 |
| 5 | 0 | 2 |
| 3 | 7 | 0 |

p 矩阵:
由 A 到 B 经过:A B
由 A 到 C 经过:A B C
由 B 到 A 经过:A B C
由 B 到 C 经过:B C
由 C 到 A 经过:A C
由 C 到 B 经过:A B C



图 7-57 有向网

求有向网中各对顶点之间最短距离的 Floyd 算法 ShortestPath_FLOYD()要求其网的邻接矩阵中对角元素的权值为 0。因为用邻接矩阵表示各顶点之间的距离,显然,同一点之

间的距离为 0。

ShortestPath_FLOYD()算法的思路很简单,首先以 2 顶点之间的直接路径为最短路径,如果能找到第 3 点,使 2 顶点通过第 3 点的路径比直接路径要短,则以这 3 点形成的路径为 2 顶点之间的最短路径。依次再找第 4 点、第 5 点、…… 如以上程序运行结果所示,根据图 7-57 及邻接矩阵,A→C 的直接距离是 11,但 A→B→C 的距离是 6,则以 6 取代 11 作为 A→C 的最短距离。

ShortestPath_FLOYD()不仅可用于有向网,也可用于无向网。因为无向网的 1 条边相当于有向网的 2 条弧。

教科书中图 7.33 是描述中国内地铁路交通的无向网。程序 algo7-11.cpp 利用 ShortestPath_FLOYD()可求得该网中每两个站点之间的最短距离。该无向网的数据是利用文件 map.txt 输入的。map.txt 的内容如下(在教科书中图 7.33 的基础上另加孤立顶点台北):

```
3
26
乌鲁木齐 呼和浩特 哈尔滨 西宁 兰州 成都 昆明 贵阳 南宁 柳州 株洲 广州 深圳 南昌 福州 上海
武汉 西安 郑州 徐州 北京 天津 沈阳 大连 长春 台北
30
乌鲁木齐 兰州 1892
呼和浩特 兰州 1145
呼和浩特 北京 668
哈尔滨 长春 242
西宁 兰州 216
兰州 西安 676
西安 成都 842
西安 郑州 511
成都 昆明 1100
成都 贵阳 967
昆明 贵阳 639
贵阳 柳州 607
柳州 株洲 672
柳州 南宁 255
贵阳 株洲 902
株洲 武汉 409
株洲 广州 675
株洲 南昌 367
广州 深圳 140
南昌 福州 622
南昌 上海 825
武汉 郑州 534
郑州 北京 695
郑州 徐州 349
徐州 天津 674
```

```
徐州 上海 651
北京 天津 137
天津 沈阳 704
沈阳 大连 397
沈阳 长春 305
```

```
// algo7-11.cpp 实现教科书图 7.33 的程序(新增孤立顶点台北)
#include"c1.h"
#include"func7-1.cpp" // 包括顶点信息类型的定义及对它的操作
#include"func7-2.cpp" // 包括弧(边)的相关信息类型的定义及对它的操作
#include"c7-1.h" // 图的数组(邻接矩阵)存储结构
#include"bo7-1.cpp" // 图的数组(邻接矩阵)存储结构的基本操作
typedef char PathMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM][MAX_VERTEX_NUM];
// 三维数组,其值只可能是 0 或 1,故用 char 类型以减少存储空间的浪费
typedef VRType DistancMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM]; // 二维数组
#include"func7-8.cpp" // 求有向网中各对顶点之间最短距离的 Floyd 算法
void path(MGraph G,PathMatrix P,int i,int j)
{ // 求由序号为 i 的起点城市到序号为 j 的终点城市最短路径沿途所经过的城市
  int k,m = i; // 起点城市序号赋给 m
  printf("依次经过的城市:\n");
    while(m!= j) // 未到终点城市
  { G.arcs[m][m].adj = INFINITY; // 对角元素赋值无穷大
    for(k = 0;k<G.vexnum;k++ )
      if(G.arcs[m][k].adj<INFINITY&&P[m][j][k])
      { // m 到 k 有直接通路,且 k 在 m 到 j 的最短路径上
        printf("%s",G.vexs[m].name);
        G.arcs[m][k].adj = G.arcs[k][m].adj = INFINITY; // 将直接通路设为不通
        m = k; // 经过的城市序号赋给 m,继续查找
        break;
      }
  }
  printf("%s\n",G.vexs[j].name); // 输出终点城市
}
void main()
{
  MGraph g;
  int i,j,k,q = 1;
  PathMatrix p; // 三维数组
  DistancMatrix d; // 二维数组
  char filename[8] = "map.txt"; // 数据文件名
  CreateFromFile(g,filename,0); // 通过文件 map.txt 构造没有相关信息的无向网 g
  for(i = 0;i<G.vexnum;i++ )
    g.arcs[i][i].adj = 0;
    // ShortestPath_FLOYD()要求对角元素值为 0,因为两点相同,其距离为 0
```

```
  ShortestPath_FLOYD(g,p,d); // 求每对顶点间的最短路径,在 func7-8.cpp 中
  while(q)
  { printf("请选择:1 查询  0 结束\n");
    scanf("%d",&q);
    if(q)
    { printf("城市代码:\n");
      for(i = 0;i<G.vexnum;i++ )
      { printf("%2d.% - 8s",i + 1,g.vexs[i].name);
        if(i%7 == 6) // 输出 7 个数据就换行
          printf("\n");
      }
      printf("\n 请输入要查询的起点城市代码 终点城市代码:");
      scanf("%d%d",&i,&j);
      if(d[i - 1][j - 1]<INFINITY) // 有通路
      { printf("%s 到 %s 的最短距离为 %d\n",g.vexs[i - 1].name,g.vexs[j - 1].name,
        d[i - 1][j - 1]);
        path(g,p,i - 1,j - 1); // 求最短路径上由起点城市到终点城市沿途所经过的城市
      }
      else
        printf("%s 到 %s 没有路径可通\n",g.vexs[i - 1].name,g.vexs[j - 1].name);
      printf("与 %s 到 %s 有关的 p 矩阵:\n",g.vexs[i - 1].name,g.vexs[j - 1].name);
      for(k = 0;k<g.vexnum;k++ )
        printf("%2d",p[i - 1][j - 1][k]);
      printf("\n");
    }
  }
}
```

**程序运行结果:**

```
请选择:1 查询  0 结束
1↙
城市代码:
 1.乌鲁木齐 2.呼和浩特 3.哈尔滨  4.西宁   5.兰州   6.成都   7.昆明
 8.贵阳    9.南宁   10.柳州  11.株洲  12.广州  13.深圳  14.南昌
15.福州   16.上海   17.武汉  18.西安  19.郑州  20.徐州  21.北京
22.天津   23.沈阳   24.大连  25.长春  26.台北
请输入要查询的起点城市代码 终点城市代码:1 10↙
乌鲁木齐到柳州的最短距离为 4694
依次经过的城市:
乌鲁木齐 兰州 西安 郑州 武汉 株洲 柳州
与乌鲁木齐到柳州有关的 p 矩阵:
 1 0 0 0 1 0 0 0 0 1 1 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0
请选择:1 查询  0 结束
1↙
```

```
城市代码：
 1.乌鲁木齐  2.呼和浩特  3.哈尔滨    4.西宁      5.兰州      6.成都      7.昆明
 8.贵阳      9.南宁     10.柳州     11.株洲     12.广州     13.深圳     14.南昌
15.福州     16.上海     17.武汉     18.西安     19.郑州     20.徐州     21.北京
22.天津     23.沈阳     24.大连     25.长春     26.台北
请输入要查询的起点城市代码 终点城市代码：21 26↙
北京到台北没有路径可通
与北京到台北有关的p矩阵：
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
请选择：1 查询   0 结束
0↙
```

和 algo7-9. cpp 中的 ShortestPath_DIJ()类似，algo7-11. cpp 中的 ShortestPath_FLOYD()也有存放最短路径通过的顶点的数组 P。在这里，数组 P 是三维的。一维数组 P[v][w][]中的信息是从顶点 v 到顶点 w 最短距离所通过的顶点。如 P[v][w][u]＝1，说明从顶点 v 到顶点 w 的最短距离通过顶点 u。而 P[v][w][t]＝0，则说明从顶点 v 到顶点 w 最短距离不通过顶点 t。以上面的程序运行结果为例，D[0][9]是乌鲁木齐到柳州的最短距离，P[0][9][]＝{1 0 0 0 1 0 0 0 0 1 1 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0}，对照序号可知，乌鲁木齐到柳州的最短距离经过乌鲁木齐、兰州、柳州、株洲、武汉、西安和郑州 7 个城市。

为了求得依次经过的城市，调用 path()。path()的算法是：对于从顶点 v 到顶点 w 最短距离路径的起点 v，它的下一点 u 是满足 G. arcs[v][u]. adj 不是无穷(v 到 u 有直接通路)同时 P[v][w][u]＝1(u 在 v 到 w 的最短路径上)条件的唯一顶点。将 G. arcs[v][u]. adj 改为无穷(避免又回头找 v)，再从 u 找满足 G. arcs[u][x]. adj 不是无穷(u 到 x 有直接通路)同时 P[v][w][x]＝1(x 在 v 到 w 的最短路径上)条件的唯一顶点 x。循环这个过程，直至到达终点 w。

孤立顶点台北和北京不在同一个连通分量中，故它们之间没有路径可通。对应的一维数组 P[20][25][]是全 0。

algo7-11. cpp 虽然能够求得任意两城市间的最短路径，但它的 DOS 界面总让我们感到不方便。我们也可以在可视化的界面下实现 algo7-11. cpp 的功能。

shortest 子目录中的软件实现了 algo7-11. cpp 的可视化。在 Visual C++6. 0 环境下打开文件 shortest\shortest. dsw，按 F7 键编译后，按 Ctrl＋F5 键运行，就会出现图 7-58 所示的界面。移动光标到待查询的起点城市顶点圆圈中并单击鼠标左键，即选定了起点城市。该城市的顶点圆圈变成虚线。再移动光标到待查询的终点城市顶点圆圈中并再次单击鼠标左键，即选定了终点城市。这时，从起点城市到终点城市最短距离的沿途城市顶点圆圈及沿线均变成虚线。同时，在信息框中还用文字说明两城市间的最短距离及依次经过的城市。

图 7-59 是运行文件 shortest\shortest. dsw 并依次用鼠标左键单击昆明和哈尔滨的结果。这个过程可以反复进行，直到按下“退出”按钮。

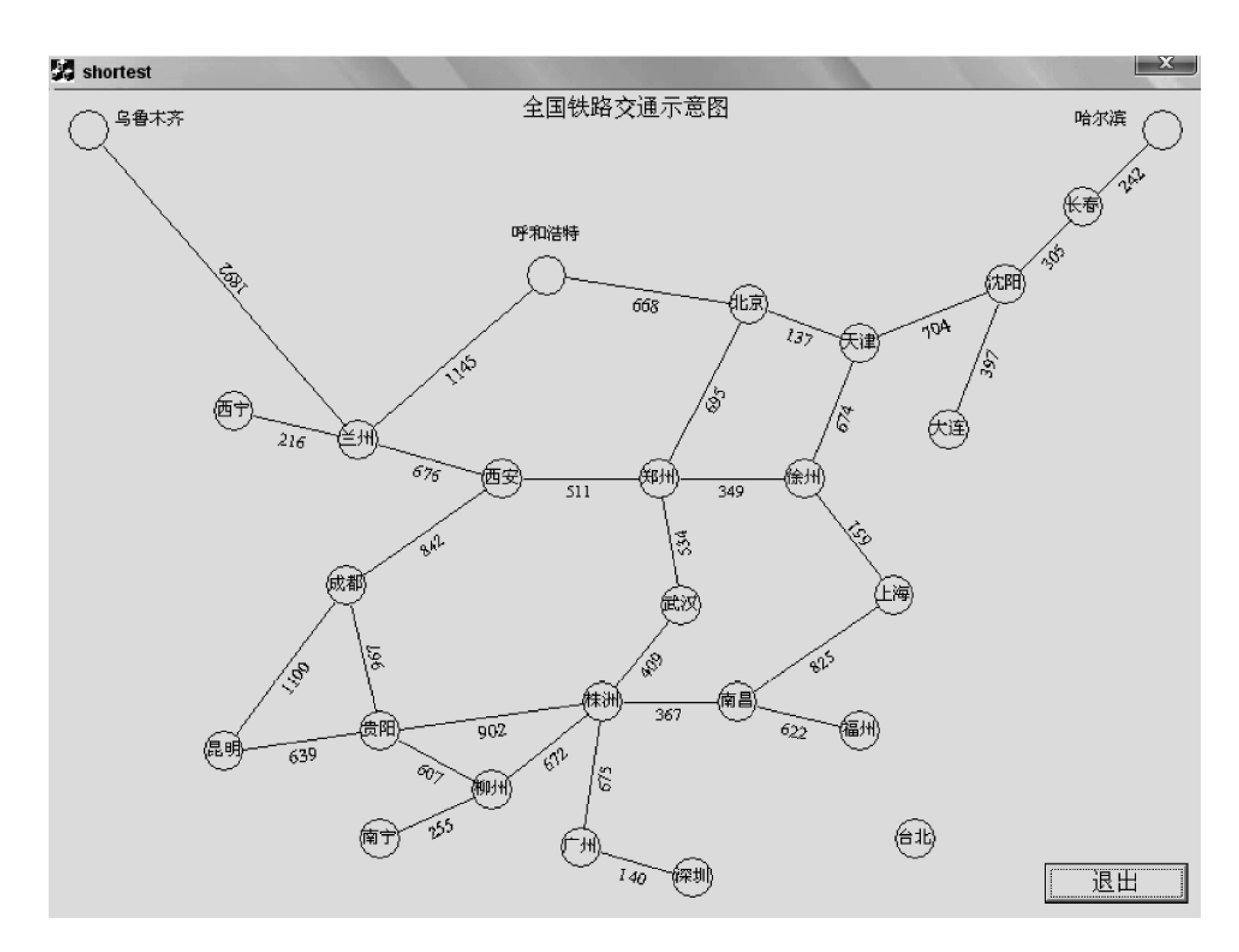



图 7-58　运行 shortest.dsw 出现的初始界面

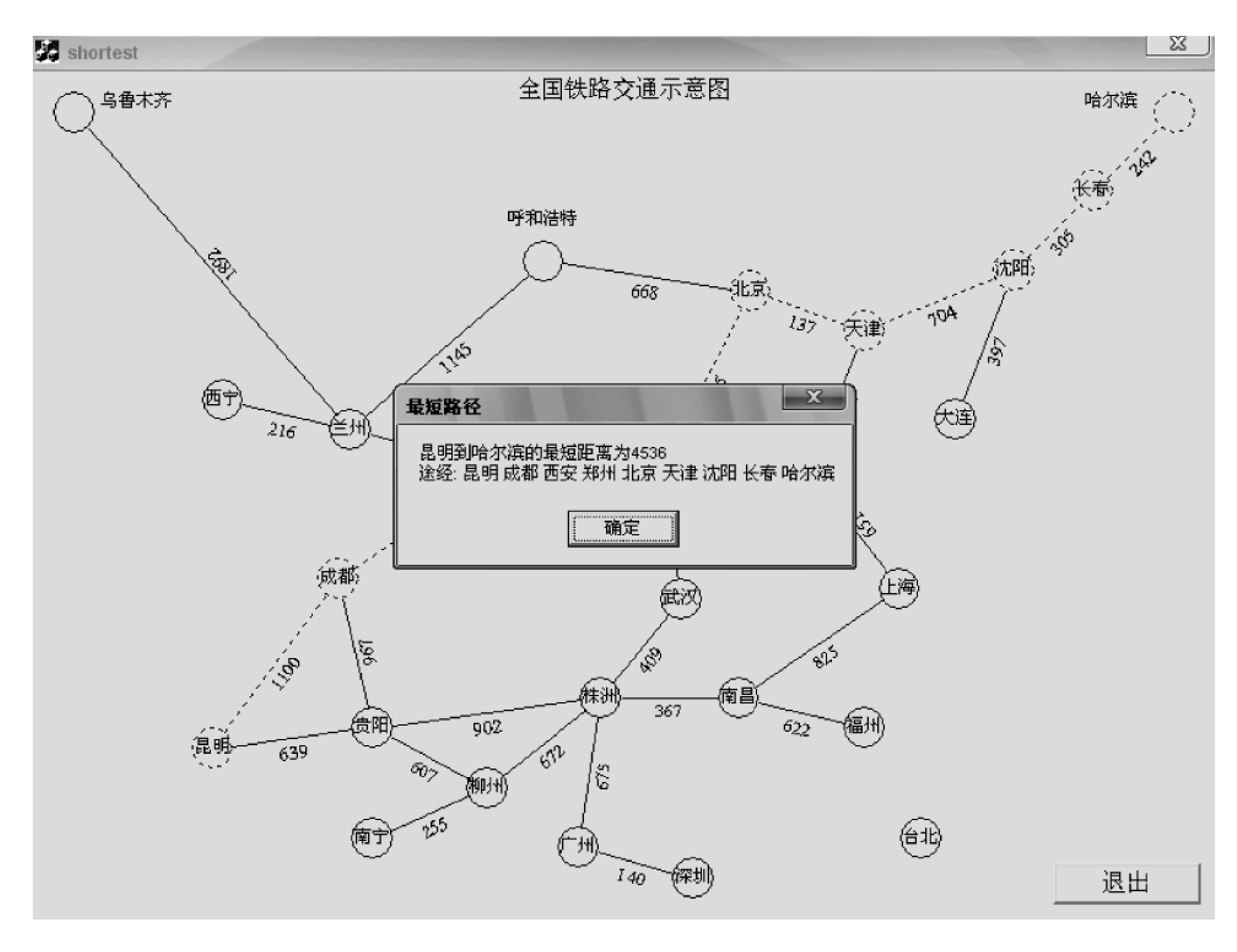



图 7-59　运行 shortest.dsw 求得昆明到哈尔滨的最短距离

algo7-11.cpp 调用的许多函数都嵌到 shortest\软件中了。有一些根据具体情况做了修改。如 mapvc.txt 的内容如下：

```
3
26
乌鲁木齐 31 26
呼和浩特 277 86
哈尔滨 607 26
西宁 109 142
兰州 175 154
成都 169 214
昆明 103 282
贵阳 187 274
南宁 187 318
柳州 247 298
株洲 307 262
广州 295 322
深圳 355 334
南昌 379 262
福州 445 274
上海 463 218
武汉 349 222
西安 253 170
郑州 337 170
徐州 415 170
北京 385 98
天津 445 114
沈阳 523 90
大连 493 150
长春 565 58
台北 475 318
30
乌鲁木齐 兰州 1892
呼和浩特 兰州 1145
…(以下同数据文件 map.txt，故略)
```

和 algo7-11.cpp 调用的数据文件 map.txt 相比，mapvc.txt 中的顶点信息不仅有城市名称，还有城市在图中的 x、y 坐标。因此，顶点信息结构如下：

```
struct VertexType // 顶点信息类型
{ char name[MAX_NAME]; // 顶点名称
  POINT pos; // 顶点坐标
};
```

其中，POINT 是 Visual C++ 6.0 软件定义的结构体，用于表示点的坐标，它的结构如下：

```
struct POINT
{ LONG x;
  LONG y;
};
```

文件 bo7-1.cpp 中的函数 CreateFromFile()和 LocateVex()、文件 func7-8.cpp 中的函数 ShortestPath_FLOYD()未做任何修改就直接用到了程序中；InputFromFile()根据 VertexType 的类型重写了，由于没有边的相关信息，InputArcFromFile()仅是空函数。

这个软件说明：在视窗时代，算法和数据结构并没有过时，仍然是软件的灵魂。视窗手段能使界面变得漂亮，但要想实现众多的复杂功能，还必须依靠算法和数据结构。

# 第 8 章

# 查　　找

在实际工作中，经常会遇到需要查找某个数据或某类数据的情况。如在学籍管理的信息中查找某人的各科成绩；在高考信息中查找总分高于分数线的考生的姓名、住址和考号等。一般要按关键字来查找，姓名、考号和总分都可以作为关键字。唯一地标识一个记录的关键字称为主关键字，如考生的考号；标识若干个记录的关键字称为次关键字，如高考分数。

一般来说，数据的值不止包括关键字，还包括其他信息。关键字只是查找的依据。为了描述方便，有时仅讨论关键字。

## 8.1　静态查找表

静态查找表在查找过程中不改变表中的数据——不插不删，故基本采用顺序存储结构。它适合用于数据不变动或不常变动的表。如高考成绩查询表、本单位职工信息表等。

### 8.1.1　顺序表的查找

```
// c8-1.h 静态查找表的顺序存储结构。在教科书第 216 页
struct SSTable // 静态查找表(见图 8-1)
{ ElemType *elem; // 数据元素存储空间基址,建表时按实际长度分配,0 号单元留空
  int length; // 表长度
};
```

SSTable　ElemType
elem
length

图 8-1　静态查找表的顺序存储结构

```
// bo8-1.cpp 静态查找表(顺序表和有序表)的基本操作(7 个),包括算法 9.1 和算法 9.2
void Creat_SeqFromFile(SSTable &ST,char* filename)
{ // 操作结果:由数据文件构造静态顺序查找表 ST(见图 8-2)
  int i;
  FILE *f; // 文件指针类型
  f = fopen(filename,"r"); // 打开数据文件,并以 f 表示
  fscanf(f,"%d",&ST.length); // 由文件输入数据元素个数
  ST.elem = (ElemType *)calloc(ST.length + 1,sizeof(ElemType));
  // 动态生成 ST.length + 1 个数据元素空间(0 号单元不用)
```

```
  if(!ST.elem) // 生成失败
    exit(OVERFLOW);
  for(i = 1;i <= ST.length;i++ )
    InputFromFile(f,ST.elem[i]);
    // 由文件依次输入静态顺序查找表 ST 的数据元素,在 func8-1.cpp 中
  fclose(f); // 关闭数据文件
}
void Ascend(SSTable &ST)
{ // 重建静态查找表为按关键字非降序排序
  int i,j,k;
  for(i = 1;i < ST.length;i++ )
  { k = i; // k 存当前关键字最小值的序号
    ST.elem[0] = ST.elem[i]; // 待比较值存[0]单元
    for(j = i + 1;j <= ST.length;j++ ) // 从元素[i]之后比起
      if LT(ST.elem[j].key,ST.elem[0].key) // 当前元素的关键字小于待比较元素的关键字
      { k = j; // 将当前元素的序号存于 k
        ST.elem[0] = ST.elem[j]; // 将当前元素的值存[0]单元
      }
    if(k != i) // 有比[i]更小的值则交换
    { ST.elem[k] = ST.elem[i];
      ST.elem[i] = ST.elem[0];
    }
  }
}
void Creat_OrdFromFile(SSTable &ST,char * filename)
{ // 操作结果:由含 n 个数据元素的数组 r 构造按关键字非降序查找表 ST
  Creat_SeqFromFile(ST,filename); // 建立无序的查找表 ST
  Ascend(ST); // 将无序的查找表 ST 重建为按关键字非降序查找表
}
Status Destroy(SSTable &ST)
{ // 初始条件:静态查找表 ST 存在。操作结果:销毁表 ST(见图 8-3)
  free(ST.elem); // 释放动态存储空间
  ST.elem = NULL; // 指针域置空
  ST.length = 0; // 表长为 0
  return OK;
}
int Search_Seq(SSTable ST,KeyType key)
{ // 在顺序表 ST 中顺序查找其主关键字等于 key 的数据元素。
  // 若找到,则返回该元素在表中的位置,否则返回 0。算法 9.1
  int i;
  ST.elem[0].key = key; // 哨兵,关键字存[0]单元
  for(i = ST.length;!EQ(ST.elem[i].key,key);--i); // 从后往前找
  return i; // 找不到时,i 为 0
}
int Search_Bin(SSTable ST,KeyType key)
```



图 8-2　具有 n 个数据元素的静态顺序查找表 ST



图 8-3　销毁表 ST

```
{ // 在有序表 ST 中折半查找其主关键字等于 key 的数据元素。
  // 若找到,则返回该元素在表中的位置,否则返回 0。算法 9.2
  int mid,low = 1,high = ST.length; // 置区间初值
  while(low<= high) // 查找范围大于 0
  { mid = (low + high)/2; // 中值
    if EQ(key,ST.elem[mid].key) // 中值是待查找元素
      return mid; // 返回其序号
    else if LT(key,ST.elem[mid].key) // 关键字小于中值
      high = mid - 1; // 继续在前半区间进行查找
    else
      low = mid + 1; // 继续在后半区间进行查找
  }
  return 0; // 顺序表中不存在待查找元素
}
void Traverse(SSTable ST,void(* Visit)(ElemType))
{ // 初始条件:静态查找表 ST 存在,Visit()是对元素操作的应用函数
  // 操作结果:按顺序对 ST 的每个元素调用函数 Visit()1 次且仅 1 次
  int i;
  ElemType * p = ++ST.elem; // p 指向第 1 个元素
  for(i = 1;i<= ST.length;i++ ) // 依次对所有元素
    Visit( * p++ ); // 调用函数 Visit(),p 指向下一个元素
}
```

bo8-1.cpp 中构造静态查找表的基本操作函数是通过数据文件构造的,在数据量较大的情况下,通过数据文件来输入数据是常用的方法,这会提高效率和降低差错。为了使 bo8-1.cpp 中的基本操作函数的应用不受具体数据元素类型限制,不在 bo8-1.cpp 中确定 ElemType 的具体类型。而是由独立的文件来定义 ElemType 和关键字的具体类型以及对它们的输入输出操作。func8-1.cpp 以教科书图 9.1 高考成绩为例定义了 ElemType 的类型。其中包括准考证号、姓名、各科成绩及总分。关键字 key 是结构体 ElemType 的一个成员,在此定义为准考证号。

```
// func8-1.cpp 包括数据元素类型的定义及对它的操作
typedef long KeyType; // 定义关键字域为长整型
#define key number // 定义关键字为准考证号
struct ElemType // 数据元素类型(以教科书图 9.1 高考成绩为例)(见图 8-4)
```

ElemType

| number | name | politics | Chinese | English | math | physics | chemistry | biology | total |
|---|---|---|---|---|---|---|---|---|---|

key

图 8-4 数据元素类型

```
{ long number; // 准考证号,与关键字类型同
  char name[9]; // 姓名(4 个汉字加 1 个串结束标志)
  int politics; // 政治
  int Chinese; // 语文
```

```
    int English; // 英语
    int math; // 数学
    int physics; // 物理
    int chemistry; // 化学
    int biology; // 生物
    int total; // 总分
};
void Visit(ElemType c) // Traverse()调用的与之配套的访问数据元素的函数
{ printf("%-8ld%-8s%4d%5d%5d%5d%5d%5d%5d%5d\n",c.number,c.name,c.politics,
    c.Chinese,c.English,c.math,c.physics,c.chemistry,c.biology,c.total);
}
void InputFromFile(FILE* f,ElemType &c) // 与之配套的从文件输入数据元素的函数
{ fscanf(f,"%ld%s%d%d%d%d%d%d%d",&c.number,c.name,&c.politics,&c.Chinese,
    &c.English,&c.math,&c.physics,&c.chemistry,&c.biology);
}
void InputKey(KeyType &k) // 与之配套的由键盘输入关键字的函数
{ scanf("%ld",&k);
}
```

```
// c8-2.h 对两个数值型关键字的比较约定为如下的宏定义。在教科书第 215 页
#define EQ(a,b)((a)==(b))
#define LT(a,b)((a)<(b))
#define LQ(a,b)((a)<=(b))
```

可以通过数据文件 f8-1.txt 构造静态表。它符合 func8-1.cpp 中从文件输入数据元素的函数对数据的要求。f8-1.txt 的内容如下：

```
5
179328 何芳芳 85 89 98 100 93 80 47
179325 陈红 85 86 88 100 92 90 45
179326 陆华 78 75 90 80 95 88 37
179327 张平 82 80 78 98 84 96 40
179324 赵小怡 76 85 94 57 77 69 44
```

顺序表中数据的排列可以是有序的，也可以是无序的。如果是有序表，可采用折半法查找，每查找一次，就把查找范围缩小一半。在数据量大的情况下，这种方法效率很高。如果是无序表，则只能从表的一端起，逐一查找了。algo8-1.cpp 是检验 bo8-1.cpp 中无序顺序表基本操作的程序。

```
// algo8-1.cpp 检验bo8-1.cpp(无序顺序表部分)的程序
#include"c1.h"
#include"func8-1.cpp" // 包括数据元素类型的定义及对它的操作
#include"c8-1.h" // 静态查找表的顺序存储结构
#include"c8-2.h" // 对两个数值型关键字比较的约定
#include"bo8-1.cpp" // 静态查找表(顺序表和有序表)的基本操作(7个)
void main()
```

```
{
  SSTable st;
  int i;
  long s;
  char filename[13]; // 存储数据文件名(包括路径)
  printf("请输入数据文件名:");
  scanf("%s",filename);
  Creat_SeqFromFile(st,filename); // 由数据文件产生顺序静态查找表 st
  for(i=1;i<=st.length;i++) // 依次计算每项数据元素的总分
    st.elem[i].total=st.elem[i].politics+st.elem[i].Chinese+st.elem[i].English+
    st.elem[i].math+st.elem[i].physics+st.elem[i].chemistry+st.elem[i].biology;
  printf("准考证号   姓名   政治 语文 外语 数学 物理 化学 生物 总分\n");
  Traverse(st,Visit); // 按顺序输出静态查找表 st
  printf("请输入待查找人的考号:");
  InputKey(s); // 由键盘输入关键字 s,在 func8-1.cpp 中
  i=Search_Seq(st,s); // 在静态查找表 st 中顺序查找含有关键字 s 的项的序号
  if(i) // 找到
    Visit(st.elem[i]); // 输出该项元素,在 func8-1.cpp 中
  else
    printf("未找到\n");
  Destroy(st); // 销毁静态查找表 st
}
```

**程序运行结果:**

```
请输入数据文件名:f8-1.txt↙
准考证号    姓名    政治   语文   外语   数学   物理   化学   生物   总分
179328     何芳芳   85     89     98     100    93     80     47     592
179325     陈红     85     86     88     100    92     90     45     586
179326     陆华     78     75     90     80     95     88     37     543
179327     张平     82     80     78     98     84     96     40     558
179324     赵小怡   76     85     94     57     77     69     44     502
请输入待查找人的考号:179326↙
179326     陆华     78     75     90     80     95     88     37     543
```

## 8.1.2 有序表的查找

algo8-2.cpp 是检验 bo8-1.cpp 中有序顺序表基本操作的程序。在它所包含的 func8-2.cpp 中,待查找的数据本身只有关键字一项。但为利用 bo8-1.cpp 中的函数,仍定义 ElemType 为结构体,其中只有一个成员,就是关键字 key。func8-2.cpp 也定义了对这种数据元素类型的输入输出操作。

```
// func8-2.cpp 包括数据元素类型的定义及对它的操作
typedef int KeyType; // 定义关键字域为整型
struct ElemType // 数据元素类型(见图 8-5)
{ KeyType key; // 仅有关键字域
```

| ElemType |
|---|
| key |

图 8-5 数据元素类型

```
};
void Visit(ElemType c) // Traverse()调用的与之配套的访问数据元素的函数
{ printf("%d",c.key);
}
void InputFromFile(FILE *f,ElemType &c) // 与之配套的从文件输入数据元素的函数
{ fscanf(f,"%d",&c.key);
}
void InputKey(KeyType &k) // 与之配套的由键盘输入关键字的函数
{ scanf("%d",&k);
}
```

数据文件 f8-2.txt 内容如下，它符合 func8-2.cpp 中从文件输入数据元素的函数对数据的要求。

```
11
5 13 19 21 37 56 64 75 80 88 92
```

```
// algo8-2.cpp 检验 bo8-1.cpp(有序表部分)的程序
#include"c1.h"
#include"func8-2.cpp" // 包括数据元素类型的定义及对它的操作
#include"c8-1.h" // 静态查找表的顺序存储结构
#include"c8-2.h" // 对两个数值型关键字比较的约定
#include"bo8-1.cpp" // 静态查找表(顺序表和有序表)的基本操作(7 个)
void main()
{
  SSTable st;
  int i;
  KeyType s;
  Creat_OrdFromFile(st,"f8-2.txt"); // 由数据文件产生非降序查找表 st(见图 8-6)
  printf("有序表为 ");
  Traverse(st,Visit); // 顺序输出静态查找表 st
  printf("\n");
  printf("请输入待查找数据的关键字：");
  InputKey(s); // 由键盘输入关键字 s，在 func8-2.cpp 中
  i = Search_Bin(st,s);
  // 在有序的静态查找表 st 中折半查找含有关键字 s 的项的序号
  if(i)
    printf("%d 是第 %d 个数据的关键字\n",st.elem[i].key,i);
  else
    printf("未找到\n");
  Destroy(st); // 销毁有序的静态查找表 st
}
```



图 8-6　具有 11 个数据元素的有序查找表 st

**程序运行结果：**

```
有序表为 5 13 19 21 37 56 64 75 80 88 92
请输入待查找数据的关键字：37↙
37 是第 5 个数据的关键字
```

### 8.1.3 静态树表的查找

静态树表是把有序的静态查找表根据数据被查找的概率生成一棵二叉树(在 c6-2.h 中定义),使得从二叉树的根结点起,查找左右子树的概率大致相等,使平均查找长度较短。若每个数据的查找概率相等,直接使用折半法即可,不必生成二叉树。

```
// func8-3.cpp 包括数据元素类型的定义及对它的操作
typedef char KeyType; // 定义关键字域为字符型
struct ElemType // 数据元素类型(见图 8-7)
{ KeyType key; // 关键字
  int weight; // 权值
};
void Visit(ElemType c) // Traverse()调用的与之配套的访问数据元素的函数
{ printf("(%c,%d)",c.key,c.weight);
}
void InputFromFile(FILE* f,ElemType &c) // 与之配套的从文件输入数据元素的函数
{ fscanf(f,"%*c%c%d",&c.key,&c.weight); // %*c吃掉回车符
}
void InputKey(KeyType &k) // 与之配套的由键盘输入关键字的函数
{ scanf("%c",&k);
}
```

| ElemType | |
|---|---|
| key | weight |

图 8-7 数据元素类型

```
// algo8-3.cpp 静态查找表(静态树表)的操作,包括算法 9.3 和算法 9.4
#include"c1.h"
#include"func8-3.cpp" // 包括数据元素类型的定义及对它的操作
#include"c8-1.h" // 静态查找表的顺序存储结构
#include"c8-2.h" // 对两个数值型关键字比较的约定
#include"bo8-1.cpp" // 静态查找表(顺序表和有序表)的基本操作(7 个)
typedef ElemType TElemType; // 定义二叉树的元素类型为数据元素类型
#include"c6-2.h" // 二叉链表的存储结构
#include"bo6-2.cpp" // 包括 InitBiTree(),DestroyBiTree()和 InOrderTraverse()函数
#define N 100 // 静态查找表的最大表长,设置 sw[]数组用到
Status SecondOptimal(BiTree &T,ElemType R[],int sw[],int low,int high)
{ // 由有序表 R[low..high]及其累计权值表 sw(其中 sw[0]==0)递归构造次优查找树 T。算法 9.3
  int j,i=low; // 有最小△Pi 值的序号,初值设为当 low==high(有序表仅 1 个元素)时的值
  double dw=sw[high]+sw[low-1]; // 教科书式 9-13 中的固定项
  double min=fabs(sw[high]-sw[low]); // △Pi 的最小值,初值设为当 low==high 时的值
  for(j=low+1;j<=high;++j) // 当 low≠high 时,选择最小的△Pi 值
    if(fabs(dw-sw[j]-sw[j-1])<min) // 找到小于 min 的值
    { i=j; // 更新有最小△Pi 值的序号
      min=fabs(dw-sw[j]-sw[j-1]); // 更新△Pi 的最小值
    }
  if(!(T=(BiTree)malloc(sizeof(BiTNode)))) // 生成结点失败
    return ERROR;
```

```
    T->data=R[i]; // 将有最小△Pi 值的数据元素赋给树结点的数据域
    if(i==low) // 有最小△Pi 值的序号是最小序号
      T->lchild=NULL; // 设左子树空
    else // 有最小△Pi 值的序号不是最小序号
      SecondOptimal(T->lchild,R,sw,low,i-1); // 递归构造次优查找左子树
    if(i==high) // 有最小△Pi 值的序号是最大序号
      T->rchild=NULL; // 设右子树空
    else // 有最小△Pi 值的序号不是最大序号
      SecondOptimal(T->rchild,R,sw,i+1,high); // 递归构造次优查找右子树
    return OK;
}
void FindSW(int sw[],SSTable ST)
{ // 按照有序表 ST 中各数据元素的 Weight 域求累计权值数组 sw,CreateSOSTree()调用
    int i;
    sw[0]=0; // 置边界值
    printf("\nsw=0 ");
    for(i=1;i<=ST.length;i++) // 由小到大计算累计权值 sw[i]并输出
    { sw[i]=sw[i-1]+ST.elem[i].weight; // 累计权值[i]=累计权值[i-1]+[i]项权值
      printf("%5d",sw[i]);
    }
}
typedef BiTree SOSTree; // 次优查找树采用二叉链表的存储结构
void CreateSOSTree(SOSTree &T,SSTable ST)
{ // 由有序表 ST 构造一棵次优查找树 T。ST 的数据元素含有权域 weight。算法 9.4
    int sw[N+1]; // 累计权值数组
    if(ST.length==0) // ST 是空表
      T=NULL; // 次优查找树 T 为空
    else // ST 不空
    { FindSW(sw,ST); // 按照有序表 ST 中各数据元素的 weight 域求累计权值表 sw
      SecondOptimal(T,ST.elem,sw,1,ST.length);
      // 由有序表 ST[1..ST.length]及其累计权值表 sw(其中 sw[0]==0)递归构造次优查找树 T
    }
}
Status Search_SOSTree(SOSTree &T,KeyType key)
{ // 在次优查找树 T 中查找主关键字等于 key 的元素。找到则返回 OK,T 指向该元素;否则返回 FALSE
    while(T) // T 非空
      if(T->data.key==key) // 树 T 根结点关键字域的值等于 key
        return OK; // 找到,返回 OK
      else if(T->data.key>key) // 树 T 根结点关键字域的值大于 key
        T=T->lchild; // T 指向 T 的左子树,继续查找
      else // 树 T 根结点关键字域的值小于 key
        T=T->rchild; // T 指向 T 的右子树,继续查找
    return FALSE; // 次优查找树 T 中不存在待查元素
}
void main()
{
    SSTable st;
```

```
    SOSTree t;
    Status i;
    KeyType s;
    Creat_OrdFromFile(st,"f8-3.txt"); // 由数据文件产生非降序静态查找表 st(见图 8-8)
    printf("        ");
    Traverse(st,Visit); // 顺序输出非降序静态查找表 st
    CreateSOSTree(t,st); // 由有序表 st 构造次优查找树 t(见图 8-9)
    printf("\n 请输入待查找的字符:");
    InputKey(s); // 由键盘输入关键字 s,在 func8-3.cpp 中
    i = Search_SOSTree(t,s); // 在次优查找树 t 中查找主关键字等于 s 的元素
    if(i) // 找到,返回 OK,t 指向该元素
      printf("%c 的权值是 %d\n",s,t->data.weight);
    else // 未找到,返回 FALSE
      printf("表中不存在此字符\n");
    DestroyBiTree(t); // 查找完毕,销毁次优查找树 t
}
```

数据文件 f8-3.txt 内容如下(以教科书例 9-1 的数据为例):

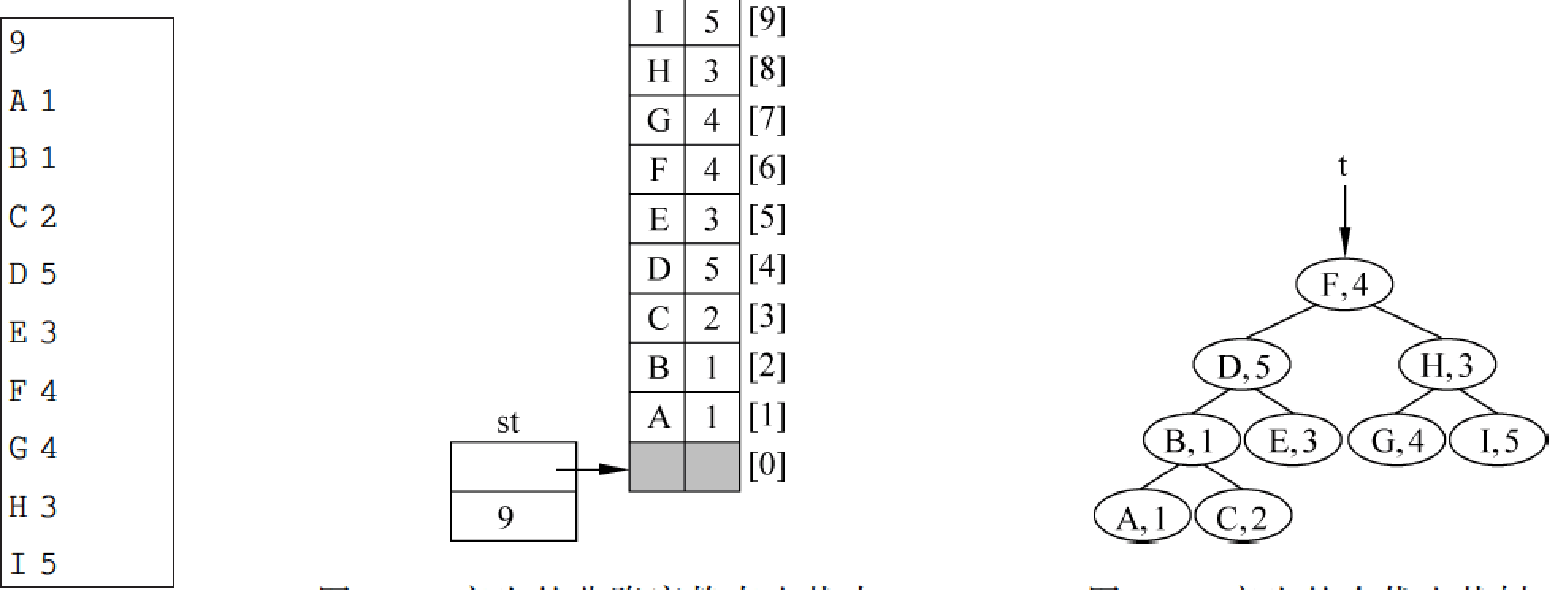



图 8-8 产生的非降序静态查找表 st　　图 8-9 产生的次优查找树 t

**程序运行结果:**

```
      (A,1)(B,1)(C,2)(D,5)(E,3)(F,4)(G,4)(H,3)(I,5)
sw = 0  1     2     4     9     12    16    20    23    28
请输入待查找的字符:G↙
G 的权值是 4
```

# 8.2 动态查找表

动态查找表在查找过程中可改变表中的数据,即可插入或删除数据,故一般采用链式存储结构。它适合用于数据经常变动的表,如飞机航班的旅客信息表等。

## 8.2.1 二叉排序树和平衡二叉树

```
// func8-4.cpp 包括算法 9.5(a)和 bo6-2.cpp,algo8-4.cpp 和 algo8-5.cpp 调用
#include"bo6-2.cpp" // 包括 InitBiTree(),DestroyBiTree()和 InOrderTraverse()函数
```

```
#define InitDSTable InitBiTree // 构造二叉排序树和平衡二叉树与初始化二叉树的操作同
#define DestroyDSTable DestroyBiTree // 销毁二叉排序树和平衡二叉树与销毁二叉树的操作同
#define TraverseDSTable InOrderTraverse
// 按关键字顺序遍历二叉排序树和平衡二叉树与中序遍历二叉树的操作同
BiTree SearchBST(BiTree T,KeyType key)
{ // 在根指针 T 所指二叉排序树或平衡二叉树中递归地查找某关键字等于 key 的数据元素，
  // 若查找成功，则返回指向该数据元素结点的指针，否则返回空指针。算法 9.5(a)
  if(!T||EQ(key,T->data.key)) // 树 T 空或待查找的关键字等于 T 所指结点的关键字
    return T; // 查找结束，返回指针 T
  else if LT(key,T->data.key) // 待查找的关键字小于 T 所指结点的关键字
    return SearchBST(T->lchild,key); // 在左子树中继续递归查找
  else // 待查找的关键字大于 T 所指结点的关键字
    return SearchBST(T->rchild,key); // 在右子树中继续递归查找
}

// bo8-2.cpp 二叉排序树的基本操作(4 个)，包括算法 9.5(b)、算法 9.6～算法 9.8
Status SearchBST(BiTree &T,KeyType key,BiTree f,BiTree &p) // 算法 9.5(b)
{ // 在根指针 T 所指二叉排序树中递归地查找其关键字等于 key 的数据元素，若查找成功，
  // 则指针 p 指向该数据元素结点，并返回 TRUE，否则指针 p 指向查找路径上访问的最后一个结点
  // 并返回 FALSE，指针 f 指向 T 的双亲，其初始调用值为 NULL
  if(!T) // 查找不成功
  { p=f; // p 指向查找路径上访问的最后一个结点
    return FALSE;
  }
  else if EQ(key,T->data.key) //   查找成功
  { p=T; // p 指向该数据元素结点
    return TRUE;
  }
  else if LT(key,T->data.key) // key 小于 T 所指结点的关键字
    return SearchBST(T->lchild,key,T,p); // 在左子树中继续递归查找
  else // key 大于 T 所指结点的关键字
    return SearchBST(T->rchild,key,T,p); //   在右子树中继续递归查找
}
Status InsertBST(BiTree &T,ElemType e)
{ // 若二叉排序树 T 中没有关键字等于 e.key 的元素，插入 e 并返回 TRUE；否则返回 FALSE。算法 9.6
  BiTree p,s;
  if(!SearchBST(T,e.key,NULL,p)) // 查找不成功，p 指向查找路径上访问的最后一个叶子结点
  { s=(BiTree)malloc(sizeof(BiTNode)); // 生成新结点
    s->data=e; // 给新结点的数据域赋值
    s->lchild=s->rchild=NULL; // 给新结点的左右孩子域赋初值空
    if(!p) // 树 T 空
      T=s; // 待插结点 *s 为新的根结点
    else if LT(e.key,p->data.key) // 树 T 不空，*s 的关键字小于 *p 的关键字
      p->lchild=s; // 待插结点 *s 为 p 所指结点的左孩子
    else // 树 T 不空，*s 的关键字大于 *p 的关键字
```

```
      p->rchild=s; // 待插结点*s为p所指结点右孩子
    return TRUE;
  }
  else
    return FALSE; // 树中已有关键字相同的结点,不再插入
}
void Delete(BiTree &p)
{ // 从二叉排序树中删除p所指结点,并重接它的左或右子树。算法9.8
  BiTree s,q=p; // q指向待删除结点(提到if语句之外)
  if(!p->rchild) // p的右子树空则只须重接它的左子树(待删结点是叶子也走此分支)(见图8-10)
  { p=p->lchild; // p指向待删除结点的左孩子
    free(q); // 释放待删除结点
  }
  else if(!p->lchild) // p的左子树空,只须重接它的右子树(见图8-11)
  { p=p->rchild; // p指向待删除结点的右孩子
```

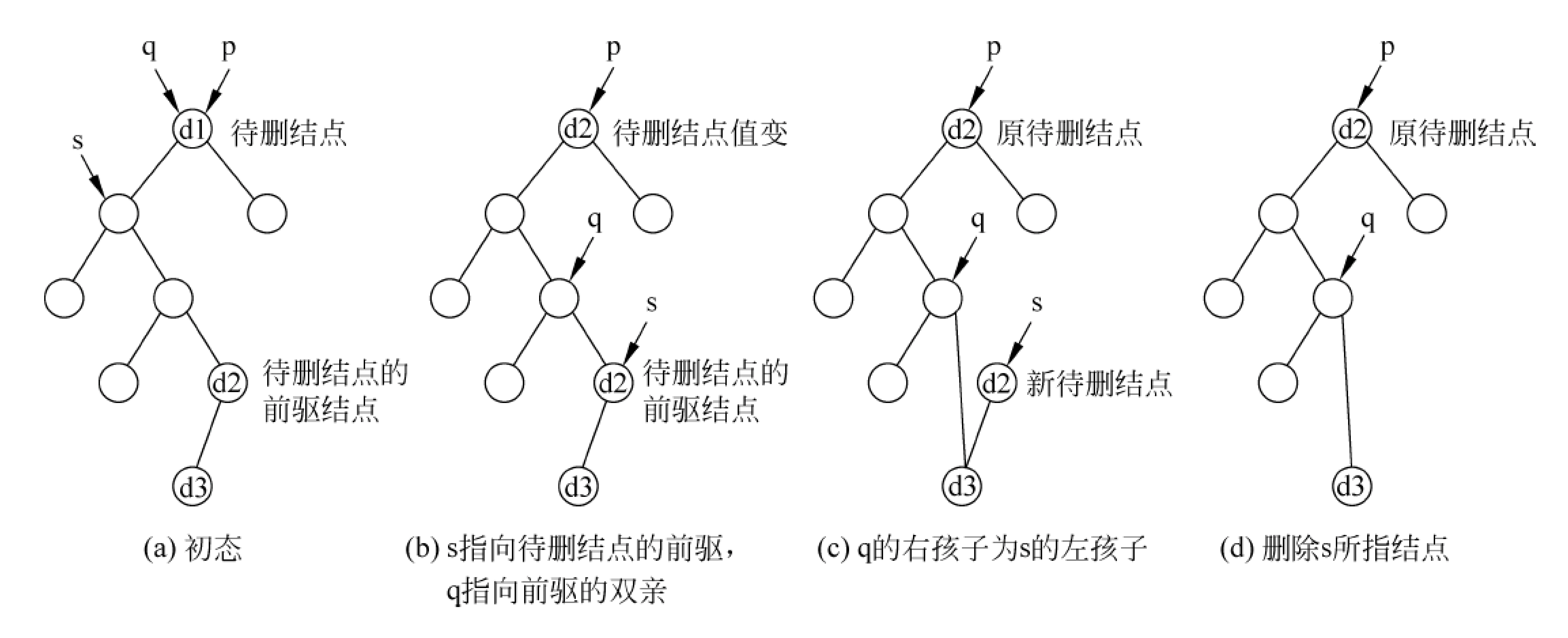
(a) 初态　(b) p指向待删结点的左孩子　(c) 释放待删结点

图 8-10　待删结点右子树空的情况

(a) 初态　(b) p指向待删结点的右孩子　(c) 释放待删结点

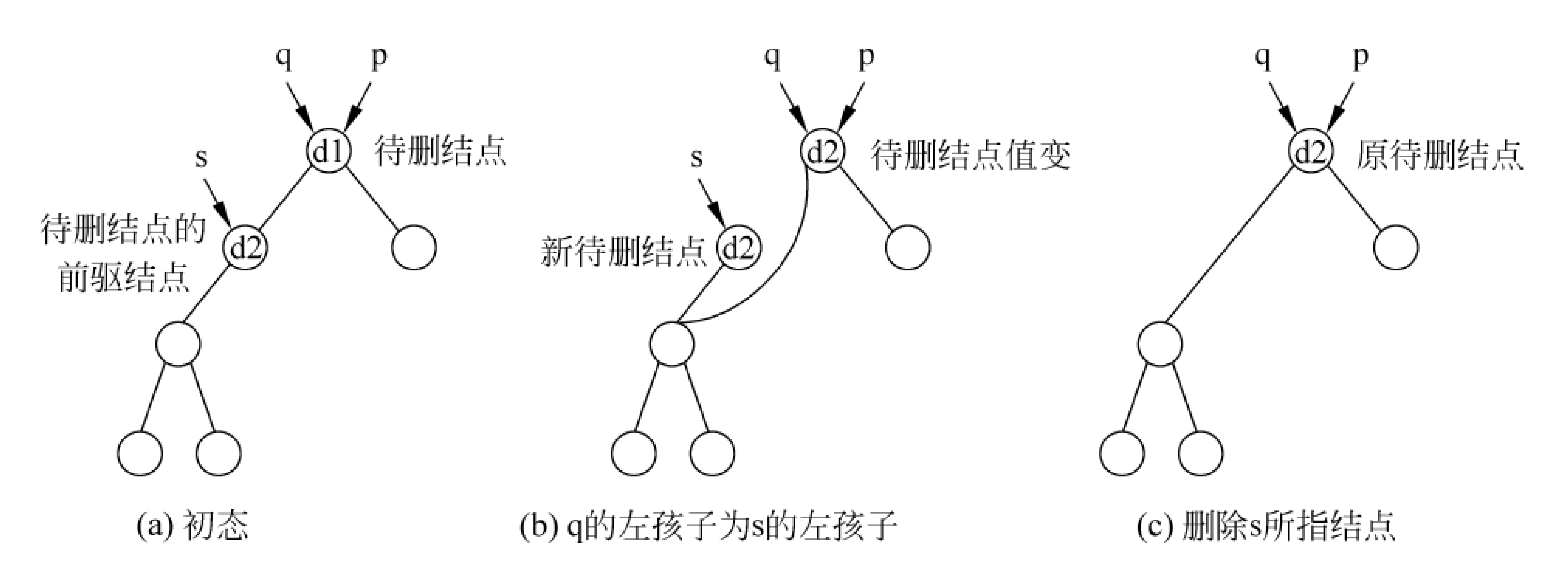
图 8-11　待删结点左子树空的情况

```
    free(q); // 释放待删除结点
  }
  else // p的左右子树均不空
  { s=p->lchild; // s指向待删除结点的左孩子(s由p转左)
    while(s->rchild) // s有右孩子
    { q=s; // q指向s
      s=s->rchild; // s指向s的右孩子
    } // s向右到尽头(s指向待删结点的前驱结点,q指向s的双亲结点)
    p->data=s->data; // 将待删结点前驱的值取代待删结点的值
    if(q!=p) // 情况①,待删结点的左孩子有右子树(见图8-12)
      q->rchild=s->lchild; // 重接*q的右子树
    else // 情况②,待删结点的左孩子没有右子树(见图8-13)
      q->lchild=s->lchild; // 重接*q的左子树
    free(s); // 释放s所指结点
  }
}
Status DeleteBST(BiTree &T,KeyType key)
{ // 若二叉排序树T中存在关键字等于key的数据元素时,则删除
  // 该数据元素结点,并返回TRUE;否则返回FALSE。算法9.7
```




(a) 初态　(b) s指向待删结点的前驱，q指向前驱的双亲　(c) q的右孩子为s的左孩子　(d) 删除s所指结点

图 8-12　情况①，待删结点的左右子树均不空，且左孩子有右子树




(a) 初态　(b) q的左孩子为s的左孩子　(c) 删除s所指结点

图 8-13　情况②，待删结点的左右子树均不空，但左孩子没有右子树

```
  if(!T) // 树 T 空
    return FALSE;
  else // 树 T 不空
  { if EQ(key,T->data.key) // 关键字等于树 T 根结点的关键字
      Delete(T); // 删除该结点
    else if LT(key,T->data.key) // 关键字小于 T 所指结点的关键字
      DeleteBST(T->lchild,key); // 在 T 的左孩子中递归查找
    else // 关键字大于 T 所指结点的关键字
      DeleteBST(T->rchild,key); // 在 T 的右孩子中递归查找
    return TRUE;
  }
}
```

```
// func8-5.cpp 包括数据元素类型的定义及对它的操作
typedef int KeyType; // 定义关键字类型为整型
struct ElemType // 数据元素类型(见图 8-14)
{ KeyType key; // 关键字
  int others; // 其他数据
};
```

ElemType

| key | others |
| --- | --- |

图 8-14　数据元素类型

```
void Visit(ElemType c) // Traverse()调用的与之配套的访问数据元素的函数
{ printf("(%d,%d)",c.key,c.others);
}
void InputFromFile(FILE *f,ElemType &c) // 与之配套的从文件输入数据元素的函数
{ fscanf(f,"%d%d",&c.key,&c.others);
}
void InputKey(KeyType &k) // 与之配套的由键盘输入关键字的函数
{ scanf("%d",&k);
}
```

```
// algo8-4.cpp 检验 bo8-2.cpp 的程序
#include"c1.h"
#include"func8-5.cpp" // 包括数据元素类型的定义及对它的操作
#include"c8-2.h" // 对两个数值型关键字比较的约定
typedef ElemType TElemType; // 定义二叉树的元素类型为数据元素类型
#include"c6-2.h" // 二叉链表的存储结构
#include"func8-4.cpp" // 包括算法 9.5(a)和 bo6-2.cpp
#include"bo8-2.cpp" // 二叉排序树的基本操作(4 个),包括算法 9.5(b)和算法 9.6~算法 9.8
void main()
{
  BiTree dt,p;
  int i,n;
  KeyType j;
  ElemType r;
  Status k;
  FILE *f; // 文件指针类型
  f=fopen("f8-4.txt","r"); // 打开数据文件 f8-4.txt
  fscanf(f,"%d",&n); // 由数据文件输入数据元素个数
  InitDSTable(dt); // 构造空二叉排序树 dt
  for(i=0;i<n;i++) // 依次在二叉排序树 dt 中插入 n 个数据元素
  { InputFromFile(f,r); // 由数据文件输入数据元素的值并赋给 r,在 func8-5.cpp 中
    k=InsertBST(dt,r); // 依次将数据元素 r 插入二叉排序树 dt 中
    if(!k) // 插入数据元素 r 失败
      printf("二叉排序树 dt 中已存在关键字为%d 的数据,故(%d,%d)无法插入到 dt 中。\n",
      r.key,r.key,r.others);
  }
  fclose(f); // 关闭数据文件
  printf("中序遍历二叉排序树 dt:\n");
  TraverseDSTable(dt,Visit); // 中序遍历二叉排序树 dt,确定 dt 是否排序
  printf("\n 先序遍历二叉排序树 dt:\n");
  PreOrderTraverse(dt,Visit); // 先序遍历二叉排序树 dt,确定 dt 的形态
  printf("\n 请输入待查找的关键字的值:");
  InputKey(j); // 由键盘输入关键字 j,在 func8-5.cpp 中
  p=SearchBST(dt,j); // 在 dt 中递归地查找关键字等于 j 的结点
  if(p) // 找到,p 指向该结点
```

```
  { printf("dt 中存在关键字为%d 的结点。",j);
    DeleteBST(dt,j); // 在 dt 中删除关键字等于 j 的结点
    printf("删除此结点后,中序遍历二叉排序树 dt:\n");
    TraverseDSTable(dt,Visit); // 中序遍历二叉排序树 dt,确定 dt 是否排序
    printf("\n 先序遍历二叉排序树 dt:\n");
    PreOrderTraverse(dt,Visit); // 先序遍历二叉排序树 dt,确定 dt 的形态
    printf("\n");
  }
  else // 未找到,p 为空
    printf("dt 中不存在关键字为%d 的结点。\n",j);
  DestroyDSTable(dt); // 销毁二叉排序树 dt
}
```

f8-4.txt 内容如下：

```
12
37 1
12 2
3 3
90 4
100 5
24 6
53 7
78 8
61 9
70 10
45 11
53 12
```

**程序运行结果：**

```
二叉排序树 dt 中已存在关键字为 53 的数据,故(53,12)无法插入到 dt 中。
中序遍历二叉排序树 dt:(见图 8-15)
(3,3)(12,2)(24,6)(37,1)(45,11)(53,7)(61,9)(70,10)(78,8)(90,4)(100,5)
先序遍历二叉排序树 dt:
(37,1)(12,2)(3,3)(24,6)(90,4)(53,7)(45,11)(78,8)(61,9)(70,10)(100,5)
请输入待查找的关键字的值:90↙
dt 中存在关键字为 90 的结点。删除此结点后,中序遍历二叉排序树 dt:(见图 8-16)
(3,3)(12,2)(24,6)(37,1)(45,11)(53,7)(61,9)(70,10)(78,8)(100,5)
先序遍历二叉排序树 dt:
(37,1)(12,2)(3,3)(24,6)(78,8)(53,7)(45,11)(61,9)(70,10)(100,5)
```

二叉排序树中任何一个结点,其左子树上所有结点的关键字值均小于该结点的关键字值;其右子树上所有结点的关键字值均大于该结点的关键字值。中序遍历二叉排序树可得到按关键字有序的序列。通过中序和先序(或中序和后序)遍历二叉排序树,就可以确定二叉排序树的形态。

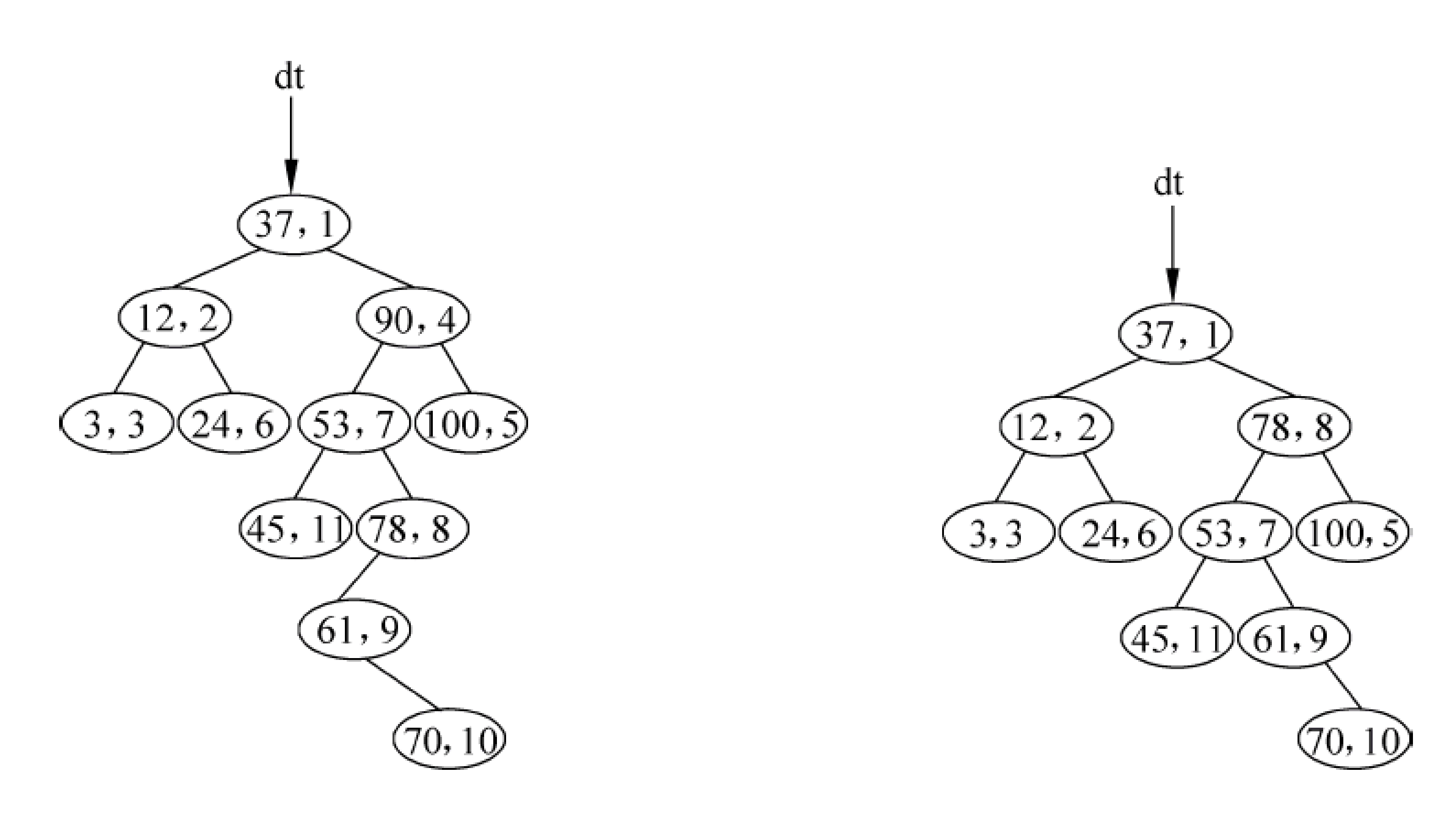



图 8-15 由文件产生的二叉排序树 dt　　图 8-16 删除 90 后的二叉排序树 dt

在二叉排序树中插入一个结点,这个结点总是叶子结点。删除一个结点从算法上分有 2 种情况:待删结点至多有一棵子树;待删结点有两棵子树。如果待删结点至多有一棵子树,则待删结点与它的双亲结点和至多存在的唯一的(左或右)孩子结点形成单链表结构,如图 8-10 和图 8-11 所示。只要将其双亲结点原来指向其的指针指向其唯一的孩子结点(若待删结点是叶子结点,则指空),就从二叉排序树中将其删除了,同时仍然保持二叉排序树的有序性。如果待删结点的两棵子树都存在,删除其需重接两棵子树,是比较麻烦的。故采用变通的方法:查找待删结点的前驱结点,这个结点是待删结点左子树的最右结点,它没有右子树。把这个结点的值赋给待删结点,相当于删除了待删结点,但却在同一位置增加了一个前驱结点,这样就有了两个相邻的前驱结点。再把原来的前驱结点删除即可。原来的前驱结点至多有一棵子树,删除它是很容易的。这样做仍保持了二叉排序树的有序性。图 8-12 和图 8-13 分别就待删结点的左孩子是否有右子树两种情况的算法做了说明。再来分析图 8-15和图 8-16 这个实际例子:图 8-15 中关键字为 90 的结点有左右两棵子树,为了删除它,首先找到它的前驱结点 78,将结点 78 的值赋给结点 90,再删除结点 78。图 8-16 显示了删除后的结果,仍然是一棵二叉排序树。当然,对称地,也可以利用待删结点的后继结点(它是待删结点右子树的最左结点)来处理这个问题。

二叉排序树的形态与数据元素插入的顺序有关。图 8-17 显示了三种由 1、2、3 三个数据元素组成的二叉排序树。当 3 个数据元素的输入顺序不同,生成了 3 棵不同的二叉排序树。

如果数据输入的顺序不当,二叉排序树的深度有可能很深,导致平均查找长度很大,失去了二叉排序树存在的意义。平衡二叉树(它也是排序树)克服了二叉排序树的这个缺点,它通过当左右子树的深度差大于 1 时调换根结点,使树的深度尽量浅,同时仍保持排序特性。

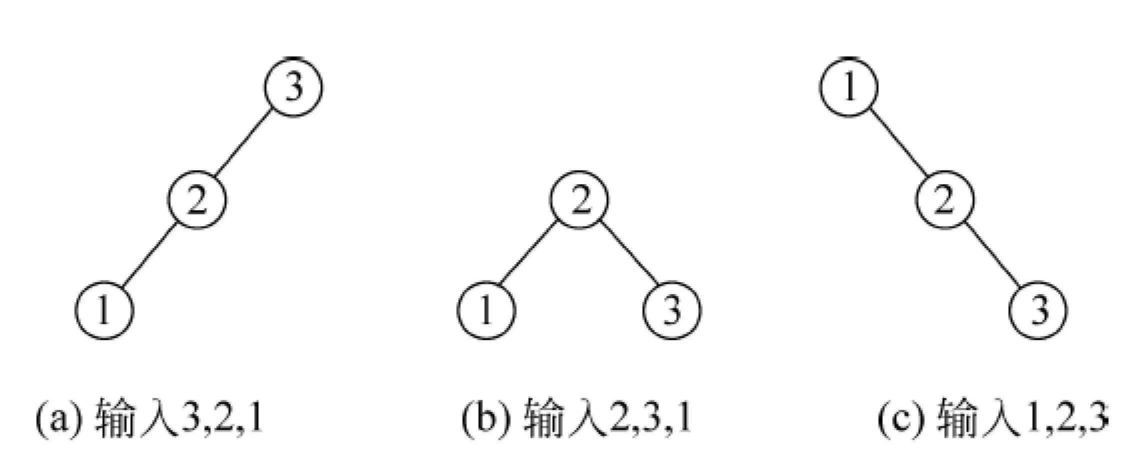



图 8-17 由 1,2,3 组成的三棵二叉排序树

```
// c8-3.h 平衡二叉树的存储结构。在教科书第 236 页(见图 8-18)
typedef struct BSTNode
```

```
{ ElemType data; // 结点的值
  int bf; // 结点的平衡因子,比二叉树结构多此项
  BSTNode * lchild, * rchild; // 左、右孩子指针
}BSTNode, * BSTree;
```

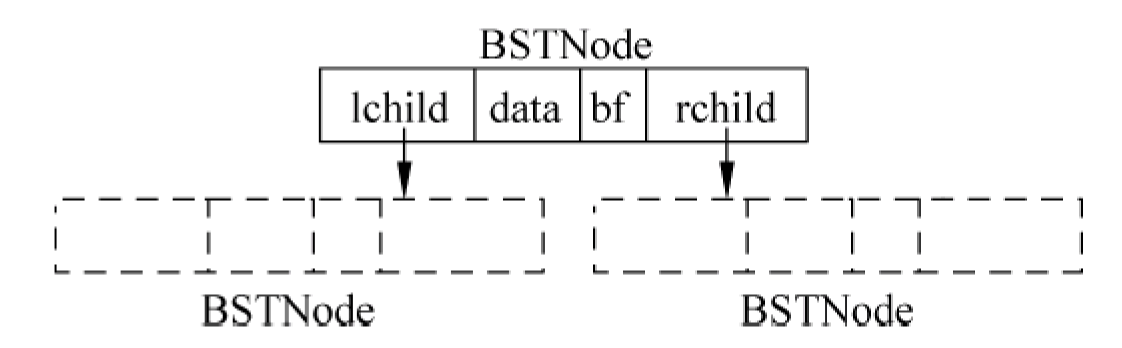



图 8-18　平衡二叉树类型存储结构

```
// bo8-3.cpp 平衡二叉树的基本操作,包括算法 9.9～算法 9.12
void R_Rotate(BSTree &p)
{ // 对以 * p 为根的二叉排序树作右旋处理,使二叉排序树的重心右移,但不修改平衡因子。
  // 处理之后 p 指向新的树根结点,即旋转处理之前的左子树的根结点。算法 9.9(见图 8-19)
  BSTree lc;
  lc = p->lchild; // lc 指向 p 的左孩子结点,lc 的左子树不变
  p->lchild = lc->rchild; // lc 的右子树挂接为 p 的左子树
  lc->rchild = p; // 原根结点成为 lc 的右孩子
  p = lc; // p 指向原左孩子结点(新的根结点)
}
void L_Rotate(BSTree &p)
{ // 对以 * p 为根的二叉排序树作左旋处理,使二叉排序树的重心左移,但不修改平衡因子。
  // 处理之后 p 指向新的树根结点,即旋转处理之前的右子树的根结点。算法 9.10(见图 8-20)
  BSTree rc;
  rc = p->rchild; // rc 指向 p 的右孩子结点,rc 的右子树不变
  p->rchild = rc->lchild; // rc 的左子树挂接为 p 的右子树
  rc->lchild = p; // 原根结点成为 rc 的左孩子
  p = rc; // p 指向原右孩子结点(新的根结点)
}
```

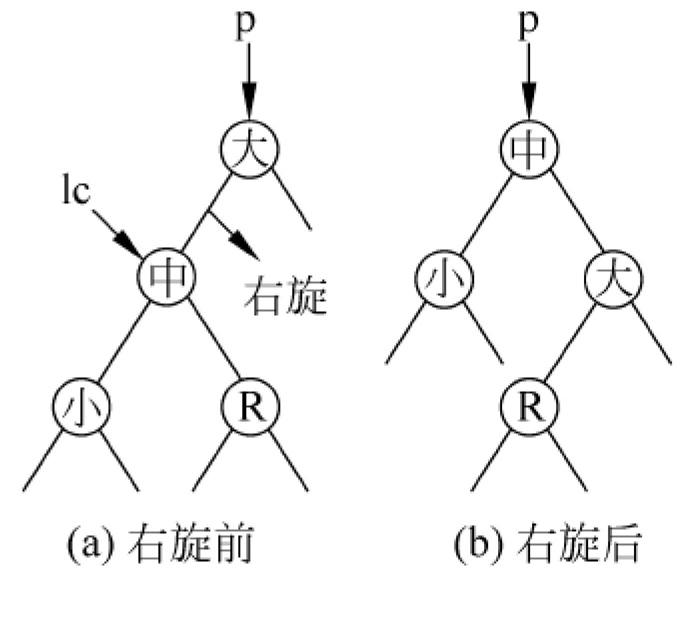



图 8-19　调用 R_Rotate()

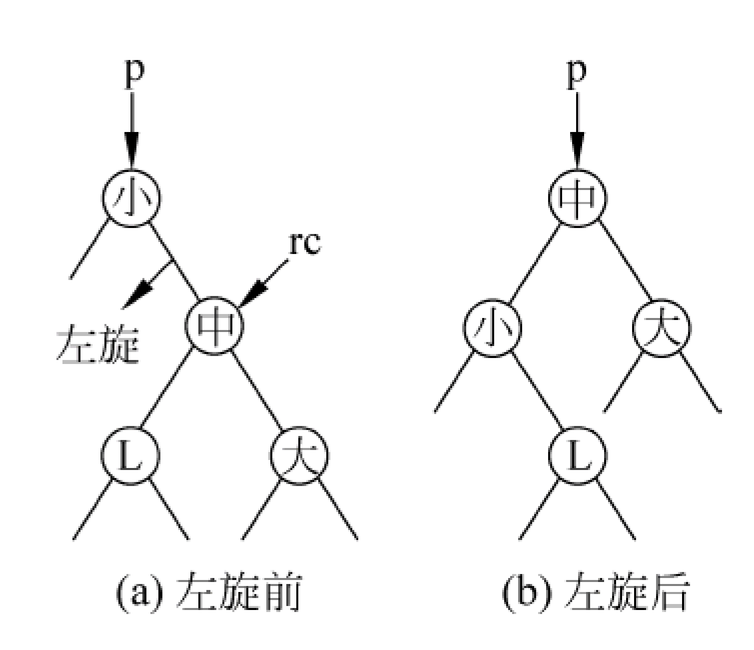



图 8-20　调用 L_Rotate()

```
void LR_Rotate(BSTree &p)
{ // 对以 * p 为根的平衡二叉树的 LR 型失衡,直接进行平衡旋转处理,不修改平衡因子
  BSTree lc = p->lchild; // lc 指向小值结点
  p->lchild = lc->rchild->rchild; // 中值结点的右子树成为大值结点的左子树
  lc->rchild->rchild = p; // 大值结点成为中值结点的右子树
```

```
  p = lc->rchild; // 根指针指向中值结点
  lc->rchild = p->lchild; // 中值结点的左子树成为小值结点的右子树
  p->lchild = lc; // 小值结点成为中值结点的左子树
}
void RL_Rotate(BSTree &p)
{ // 对以 * p 为根的平衡二叉树的 RL 型失衡,直接进行平衡旋转处理,不修改平衡因子
  BSTree rc = p->rchild; // rc 指向大值结点
  p->rchild = rc->lchild->lchild; // 中值结点的左子树成为小值结点的右子树
  rc->lchild->lchild = p; // 小值结点成为中值结点的左子树
  p = rc->lchild; // 根指针指向中值结点
  rc->lchild = p->rchild; // 中值结点的右子树成为大值结点的左子树
  p->rchild = rc; // 大值结点成为中值结点的右子树
}
#define LH +1 // 左高。在教科书第 236 页
#define EH  0 // 等高
#define RH -1 // 右高
void LeftBalance(BSTree &T) // 算法 9.12
{ // 初始条件:原本平衡二叉排序树 T 的左子树比右子树高(T->bf = 1),
  //          又在左子树中插入了结点,并导致左子树更高,破坏了树 T 的平衡性
  // 操作结果:对不平衡的树 T 作左平衡旋转处理,使树 T 的重心右移,
  //          实现新的平衡。本算法结束时,T 指向新的平衡二叉树根结点
  BSTree lc,rd;
  lc = T->lchild; // lc 指向 * T 的左孩子结点
  switch(lc->bf) // 检查 * T 的左子树的平衡度,并作相应平衡处理
  { case LH: // 新结点插入在 * T 的左孩子的左子树上,导致左子树的平衡因子为左高,
          // 形成 LL(/)型不平衡,要作单右旋处理(见图 8-21)
          T->bf = lc->bf = EH; // 旋转后,原根结点和左孩子结点(新根结点)的平衡因子都为 0
          R_Rotate(T); // 对树 T 作右旋处理,使树 T 的重心右移
          break;
    case RH: // 新结点插入在 * T 的左孩子的右子树上,导致左子树的平衡因子为右高,
          // 形成 LR(<)型不平衡,要作双旋处理(见图 8-22(a))
          rd = lc->rchild; // rd 指向 * T 的左孩子的右子树根结点
          switch(rd->bf) // 检查 * T 的左孩子的右子树的平衡度,修改 * T 及其左孩子的平衡因子
```

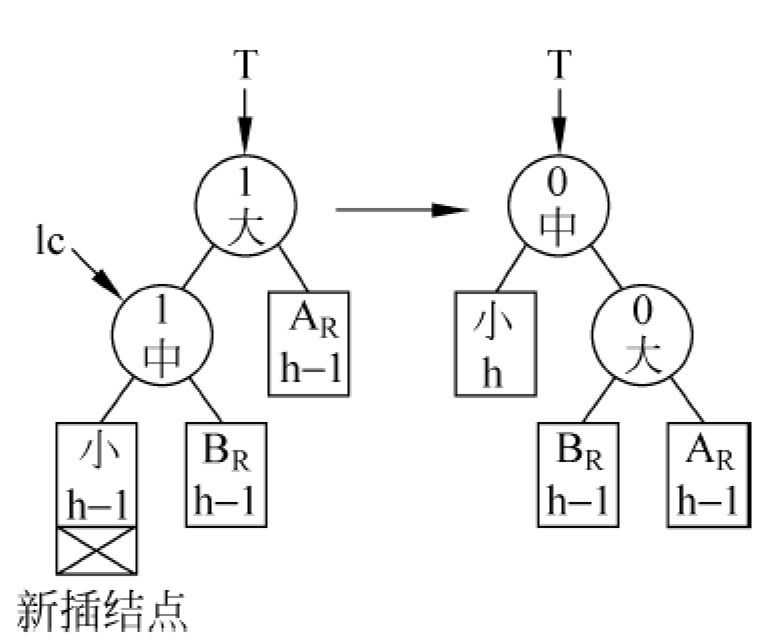



图 8-21 LL 型平衡旋转

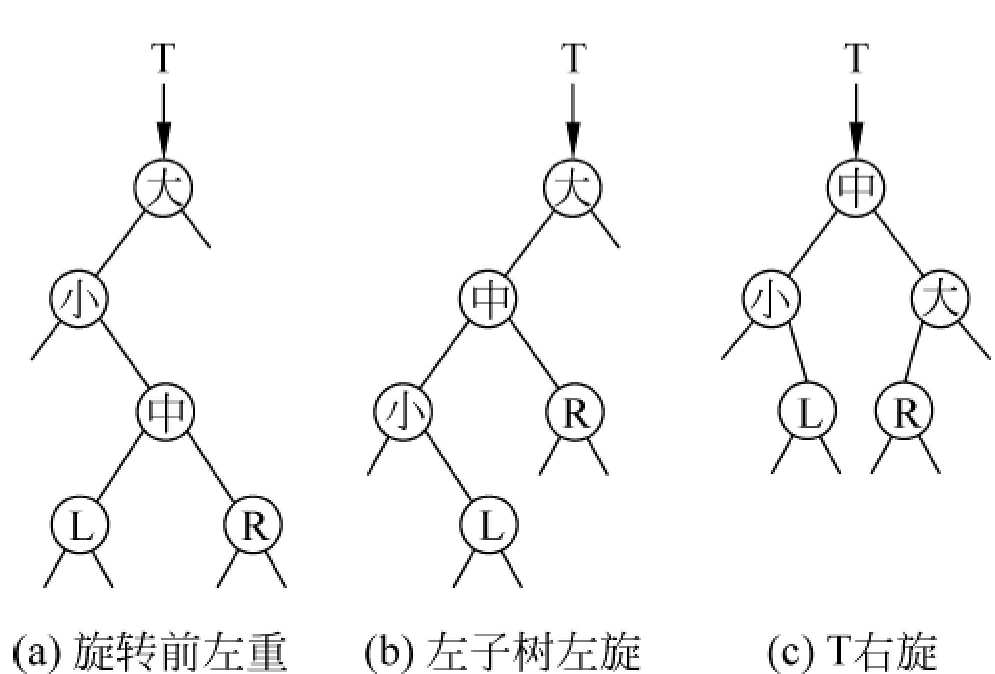



图 8-22 LR 型平衡旋转

```
{ case LH: // 新结点插入在 * T 的左孩子的右子树的左子树上(情况①,见图 8-23(a))
          T->bf = RH; // 旋转后,原根结点的平衡因子为右高
          lc->bf = EH; // 旋转后,原根结点的左孩子结点平衡因子为等高
          break;
  case EH: // 新结点插入为 * T 的左孩子的右孩子(叶子)(情况②,见图 8-23(b))
          T->bf = lc->bf = EH; // 旋转后,原根和左孩子结点的平衡因子都为等高
          break;
  case RH: // 新结点插入在 * T 的左孩子的右子树的右子树上(情况③,见图 8-23(c))
          T->bf = EH; // 旋转后,原根结点的平衡因子为等高
          lc->bf = LH; // 旋转后,原根结点的左孩子结点平衡因子为左高
}
```

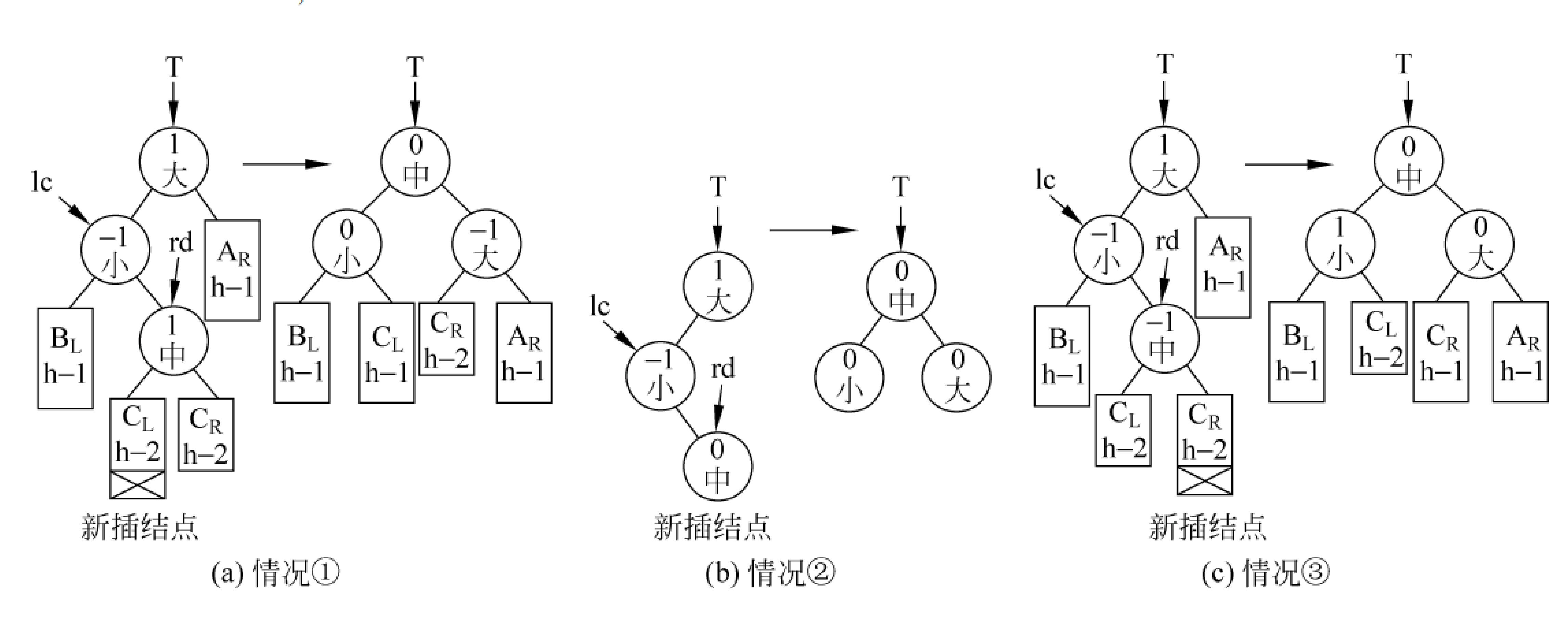



图 8-23　LR 型平衡旋转 3 种情况平衡因子变化

```
          rd->bf = EH; // 旋转后的新根结点( * T 的左孩子的右孩子结点)的平衡因子为等高
#ifndef FLAG // 没定义 FLAG,使用 2 个函数实现双旋处理,同教科书
          L_Rotate(T->lchild);
          // 对 * T 的左子树作左旋处理,使 * T 成为 LL(/)型不平衡(见图 8-22(b))
          R_Rotate(T); // 对 * T 作右旋处理(见图 8-22(c))
#else // 定义了 FLAG,直接处理 LR 型不平衡
          LR_Rotate(T); // 对 * T 直接进行平衡旋转处理
#endif
  }
}
void RightBalance(BSTree &T)
{ // 初始条件:原本平衡二叉排序树 T 的右子树比左子树高(T->bf =- 1),又在右子树中插入了结点,
  //          并导致右子树更高,破坏了树 T 的平衡性
  // 操作结果:对不平衡的树 T 作右平衡旋转处理,使树 T 的重心左移,实现新的平衡。
  //          本算法结束时,T 指向新的平衡二叉树根结点
  BSTree rc,ld;
  rc = T->rchild; // rc 指向 * T 的右孩子结点
  switch(rc->bf) // 检查 * T 的右子树的平衡度,并作相应平衡处理
  { case RH: // 新结点插入在 * T 的右孩子的右子树上,导致右子树的平衡因子为右高,
          // 形成 RR(\)型不平衡,要作单左旋处理(见图 8-24)
          T->bf = rc->bf = EH; // 旋转后,原根结点和右孩子结点(新根结点)的平衡因子都为 0
```

```
            L_Rotate(T); // 对树 T 作左旋处理,使树 T 的重心左移
            break;
    case LH: // 新结点插入在 * T 的右孩子的左子树上,导致右子树的平衡因子为左高,
             // 形成 RL(>)型不平衡,要作双旋处理(见图 8-25(a))
```

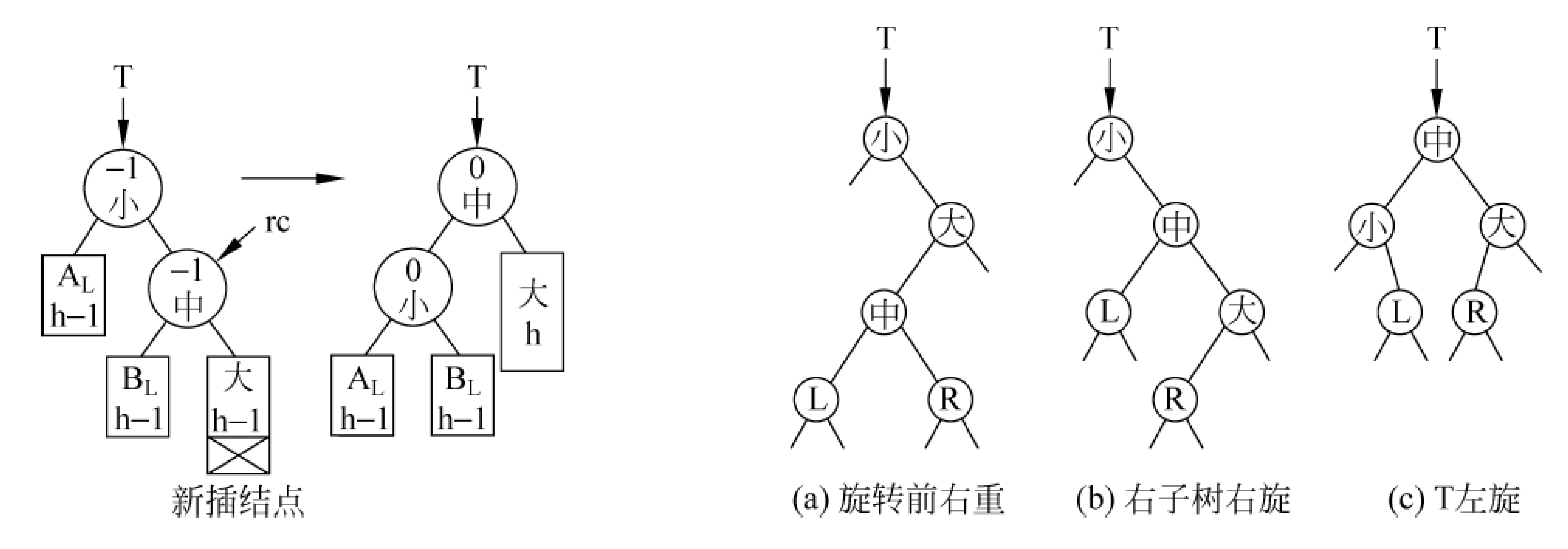



图 8-24 RR 型平衡旋转　　图 8-25 RL 型平衡旋转

```
            ld = rc->lchild; // ld 指向 * T 的右孩子的左子树根结点
            switch(ld->bf) // 检查 * T 的右孩子的左子树的平衡度,修改 * T 及其右孩子的平衡因子
            { case RH: // 新结点插入在 * T 的右孩子的左子树的右子树上(情况①,见图 8-26(a))
                       T->bf = LH; // 旋转后,原根结点的平衡因子为左高
                       rc->bf = EH; // 旋转后,原根结点的右孩子结点平衡因子为等高
                       break;
              case EH: // 新结点插入为 * T 的右孩子的左孩子(叶子)(情况②,见图 8-26(b))
                       T->bf = rc->bf = EH; // 旋转后,原根和右孩子结点的平衡因子都为等高
                       break;
              case LH: // 新结点插入在 * T 的右孩子的左子树的左子树上(情况③,见图 8-26(c))
                       T->bf = EH; // 旋转后,原根结点的平衡因子为等高
                       rc->bf = RH; // 旋转后,原根结点的右孩子结点的平衡因子为右高
            }
```

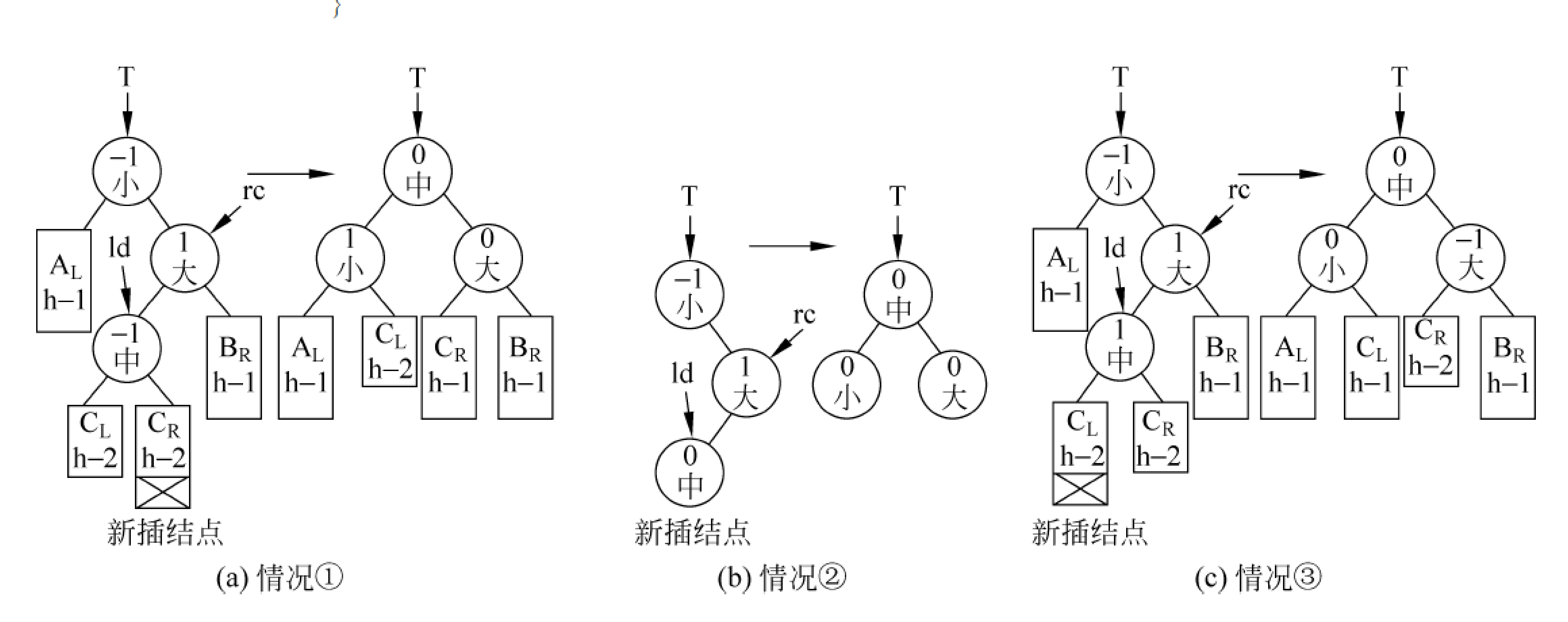



图 8-26 RL 型平衡旋转 3 种情况平衡因子变化

```
            ld->bf = EH; // 旋转后的新根结点( * T 的右孩子的左孩子结点)的平衡因子为等高
#ifndef FLAG // 没定义 FLAG,使用 2 个函数实现双旋处理
            R_Rotate(T->rchild);
            // 对 * T 的右子树作右旋处理,使 * T 成为 RR(\)型不平衡(见图 8-25(b))
```

```
            L_Rotate(T); // 对 * T 作左旋处理(见图 8-25(c))
#else // 定义了 FLAG,直接处理 RL 型不平衡
            RL_Rotate(T); // 对 * T 直接进行平衡旋转处理
#endif
  }
}
Status InsertAVL(BSTree &T,ElemType e,Boolean &taller)
{ // 若在平衡的二叉排序树 T 中不存在和 e 有相同关键字的结点,则插入一个数据元素为 e 的新结点,
  // 并返回 TRUE;否则返回 FALSE。若因插入而使二叉排序树 T 失去平衡,则作平衡旋转处理,
  // 布尔变量 taller 反映在调用 InsertAVL()前后,T 是否长高。算法 9.11
  if(!T) // 树或子树 T 空
  { // 插入新结点,树"长高",置 taller 为 TRUE
    T = (BSTree)malloc(sizeof(BSTNode)); // 生成新结点,且 T 指向其
    T->data = e; // 给新结点赋值
    T->lchild = T->rchild = NULL; // 新结点是叶子结点
    T->bf = EH; // 叶子结点的平衡因子为等高
    taller = TRUE; // 以 T 为根结点的树长高了(深度由 0 变为 1),此信息返回给 T 的双亲结点
  }
  else // 树或子树 T 不空
  { if EQ(e.key,T->data.key) // T 所指结点的关键字和 e 相同,不再插入
    { // taller = FALSE; // 树 T 保持原来的形态,没有长高(此句可省)
      return FALSE; // 没有在树 T 中插入结点的标志
    }
    if LT(e.key,T->data.key) // e 的关键字小于 * T 的关键字
    { if(!InsertAVL(T->lchild,e,taller)) // 继续在 * T 的左子树中递归调用 InsertAVL()
        return FALSE; // 如未插入 e,返回没有在树 T 中插入结点的标志
      if(taller) // 如已将 e 插入到 * T 的左子树中且左子树"长高"
        switch(T->bf) // 检查 * T 的平衡度,并作适当处理
        { case LH: // 原本树 T 的左子树比右子树高,现在 T 的左子树又"长高"了
                  LeftBalance(T); // 对树 T 作左平衡处理,使树 T 的重心右移
                  taller = FALSE; // 对 T 作左平衡处理后,树 T 的深度同插入结点 e 前,未长高
                  break;
          case EH: // 原本树 T 的左、右子树等高,现在 T 的左子树又"长高"了
                  T->bf = LH; // T 的平衡因子由 0 变为 1(左高)
                  taller = TRUE; // 树 T 比插入结点 e 前"长高"了,此信息返回给 T 的双亲结点
                  break;
          case RH: T->bf = EH; // 原本树 T 的右子树比左子树高,现在 T 的左、右子树等高
                  taller = FALSE; // 树 T 没有长高,此信息返回给 T 的双亲结点
        }
    }
    else // e 的关键字大于 * T 的关键字
    { if(!InsertAVL(T->rchild,e,taller)) // 继续在 * T 的右子树中进行搜索,如未插入
        return FALSE; // 没有在树 T 中插入结点的标志
      if(taller) // 已插入到 T 的右子树且右子树"长高"
```

```
      switch(T->bf) // 检查 T 的平衡度
      { case LH: T->bf = EH; // 原本树 T 的左子树比右子树高,现在 T 的左、右子树等高
                 taller = FALSE; // 树 T 没有长高,此信息返回给 T 的双亲结点
                 break;
        case EH: // 原本树 T 的左、右子树等高,现在 T 的右子树又"长高"了
                 T->bf = RH; // T 的平衡因子由 0 变为 -1(右高)
                 taller = TRUE; // 树 T 比插入结点 e 前"长高"了,此信息返回给 T 的双亲结点
                 break;
        case RH: // 原本右子树比左子树高,现在 T 的右子树又"长高"了
                 RightBalance(T); // 对树 T 作右平衡处理,使树 T 的重心左移
                 taller = FALSE; // 对 T 作右平衡处理后,树 T 的深度同插入结点 e 前,未长高
      }
    }
  }
  return TRUE;
}
```

```
// algo8-5.cpp 检验 bo8-3.cpp 的程序
#include"c1.h"
#include"func8-5.cpp" // 包括数据元素类型的定义及对它的操作
#include"c8-2.h" // 对两个数值型关键字比较的约定
typedef ElemType TElemType; // 定义二叉树的元素类型为数据元素类型
#include"c8-3.h" // 平衡二叉树的存储结构
typedef BSTree BiTree; // 定义二叉树的指针类型为平衡二叉树的指针类型
#include"func8-4.cpp" // 包括算法 9.5(a)和 bo6-11.cpp
//#define FLAG // 加此句在 bo8-3.cpp 中直接做 RL 和 LR 平衡处理。第 9 行
#include"bo8-3.cpp" // 平衡二叉树的基本操作
void main()
{
  BSTree dt,p;
  int i,n;
  KeyType j;
  ElemType r;
  Boolean flag;
  Status k;
  FILE *f; // 文件指针类型
  f = fopen("f8-4.txt","r"); // 打开数据文件 f8-4.txt
  fscanf(f,"%d",&n); // 由数据文件输入数据元素个数
  InitDSTable(dt); // 初始化空平衡二叉树 dt
  for(i = 0;i<n;i++) // 依次在平衡二叉树 dt 中插入 n 个数据元素
  { InputFromFile(f,r); // 由数据文件输入数据元素的值并赋给 r,在 func8-5.cpp 中
    k = InsertAVL(dt,r,flag); // 在平衡二叉树 dt 中插入数据元素 r,使仍为平衡二叉树
    if(k) // 插入数据元素 r 成功
    { printf("插入关键字为%d 的结点后,按关键字顺序(中序)遍历平衡二叉树 dt\n",r.key);
      TraverseDSTable(dt,Visit); // 按关键字顺序(中序)遍历二叉树 dt,确定 dt 是否排序
```

```
      printf("\n 先序遍历平衡二叉树 dt\n");
      PreOrderTraverse(dt,Visit); // 先序遍历平衡二叉树 dt,确定 dt 的形态
      printf("\n");
    }
    else // 插入数据元素 r 失败
      printf("平衡二叉树 dt 中已存在关键字为 %d 的数据,故(%d, %d)无法插入到 dt 中。\n",
      r.key,r.key,r.others);
  }
  fclose(f); // 关闭数据文件
  printf("请输入待查找的关键字的值:");
  InputKey(j); // 由键盘输入关键字 j,在 func8-5.cpp 中
  p = SearchBST(dt,j); // 在平衡二叉树 dt 中递归地查找关键字等于 j 的结点
  if(p) // 找到,p 指向该结点
    printf("dt 中存在关键字为 %d 的结点,其值为(%d, %d)。\n",j,p->data.key,
    p->data.others);
  else
    printf("dt 中不存在关键字为 %d 的结点。\n",j);
  DestroyDSTable(dt); // 销毁平衡二叉树 dt
}
```

**程序运行结果：**

```
插入关键字为 37 的结点后,按关键字顺序(中序)遍历平衡二叉树 dt(见图 8-27)
(37,1)
先序遍历平衡二叉树 dt
(37,1)
插入关键字为 12 的结点后,按关键字顺序(中序)遍历平衡二叉树 dt(见图 8-28)
(12,2)(37,1)
先序遍历平衡二叉树 dt
(37,1)(12,2)
插入关键字为 3 的结点后,按关键字顺序(中序)遍历平衡二叉树 dt(见图 8-29)
```

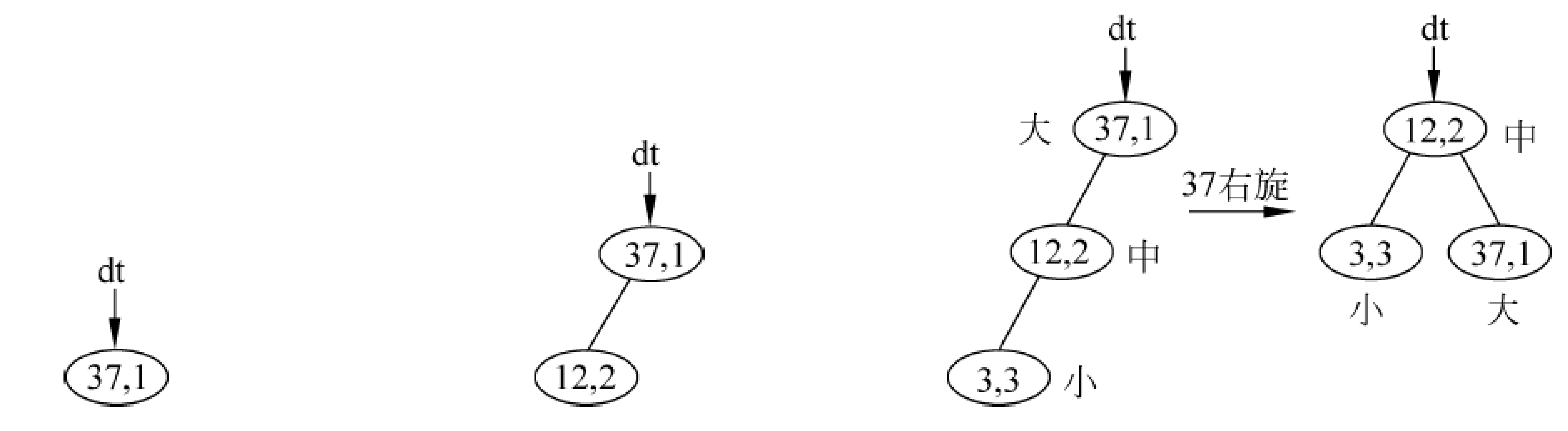



图 8-27　插入结点 37　　图 8-28　插入结点 12　　图 8-29　插入结点 3,LL 型不平衡

```
(3,3)(12,2)(37,1)
先序遍历平衡二叉树 dt
(12,2)(3,3)(37,1)
插入关键字为 90 的结点后,按关键字顺序(中序)遍历平衡二叉树 dt(见图 8-30)
(3,3)(12,2)(37,1)(90,4)
先序遍历平衡二叉树 dt
(12,2)(3,3)(37,1)(90,4)
```

插入关键字为 100 的结点后,按关键字顺序(中序)遍历平衡二叉树 dt(见图 8-31)
(3,3)(12,2)(37,1)(90,4)(100,5)

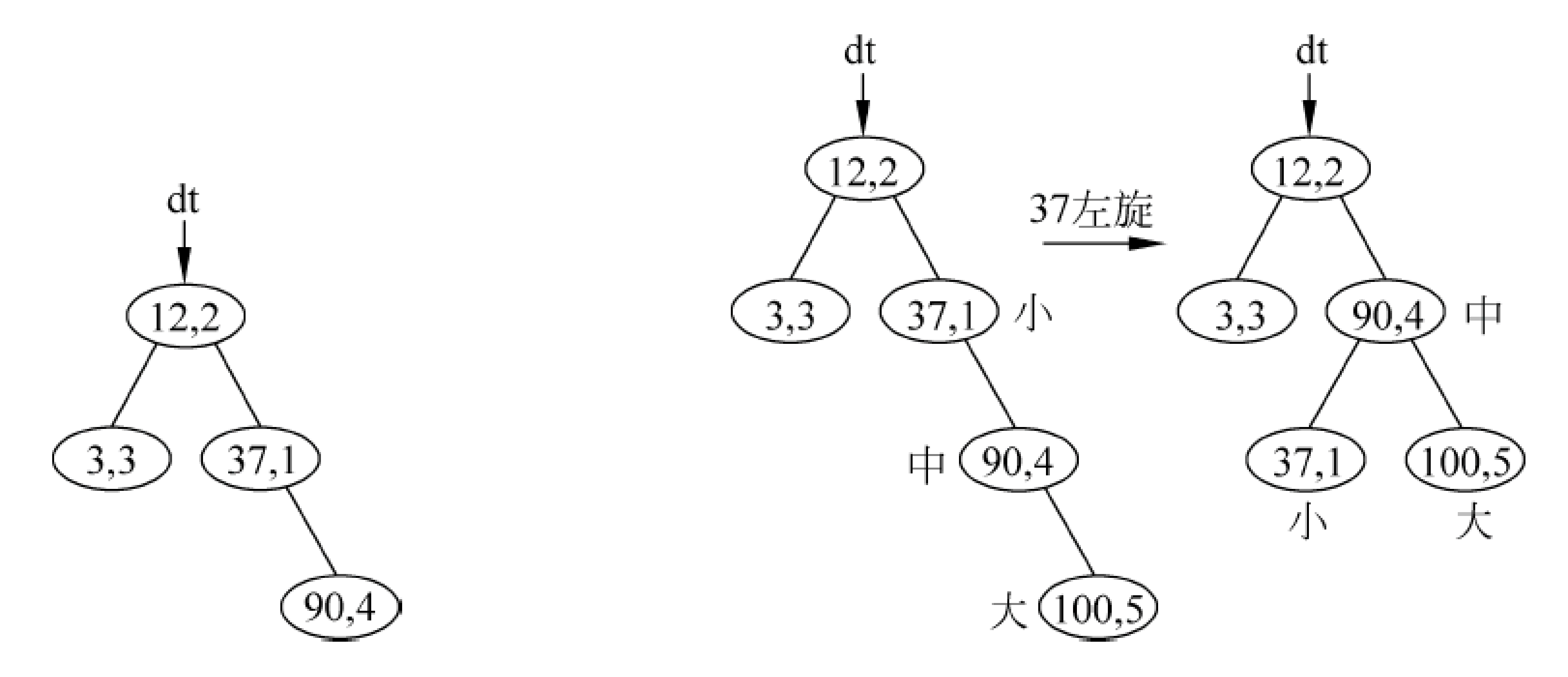



图 8-30 插入结点 90　　图 8-31 插入结点 100,RR 型不平衡

先序遍历平衡二叉树 dt
(12,2)(3,3)(90,4)(37,1)(100,5)
插入关键字为 24 的结点后,按关键字顺序(中序)遍历平衡二叉树 dt(见图 8-32)
(3,3)(12,2)(24,6)(37,1)(90,4)(100,5)

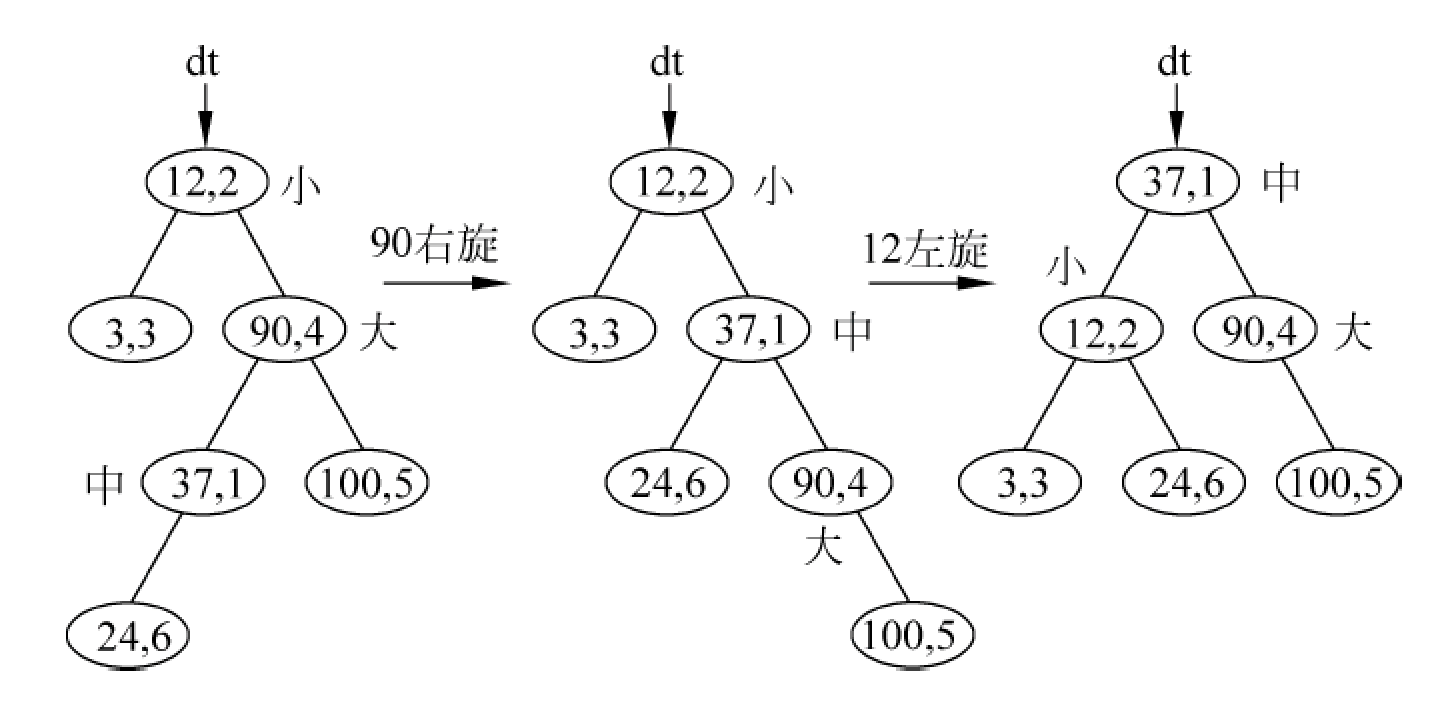



图 8-32 插入结点 24,RL 型不平衡

先序遍历平衡二叉树 dt
(37,1)(12,2)(3,3)(24,6)(90,4)(100,5)
插入关键字为 53 的结点后,按关键字顺序(中序)遍历平衡二叉树 dt(见图 8-33)
(3,3)(12,2)(24,6)(37,1)(53,7)(90,4)(100,5)
先序遍历平衡二叉树 dt
(37,1)(12,2)(3,3)(24,6)(90,4)(53,7)(100,5)
插入关键字为 78 的结点后,按关键字顺序(中序)遍历平衡二叉树 dt
(3,3)(12,2)(24,6)(37,1)(53,7)(78,8)(90,4)(100,5)
先序遍历平衡二叉树 dt(见图 8-34)
(37,1)(12,2)(3,3)(24,6)(90,4)(53,7)(78,8)(100,5)
插入关键字为 61 的结点后,按关键字顺序(中序)遍历平衡二叉树 dt
(3,3)(12,2)(24,6)(37,1)(53,7)(61,9)(78,8)(90,4)(100,5)
先序遍历平衡二叉树 dt(见图 8-35)
(37,1)(12,2)(3,3)(24,6)(90,4)(61,9)(53,7)(78,8)(100,5)
插入关键字为 70 的结点后,按关键字顺序(中序)遍历平衡二叉树 dt(见图 8-36)
(3,3)(12,2)(24,6)(37,1)(53,7)(61,9)(70,10)(78,8)(90,4)(100,5)
先序遍历平衡二叉树 dt

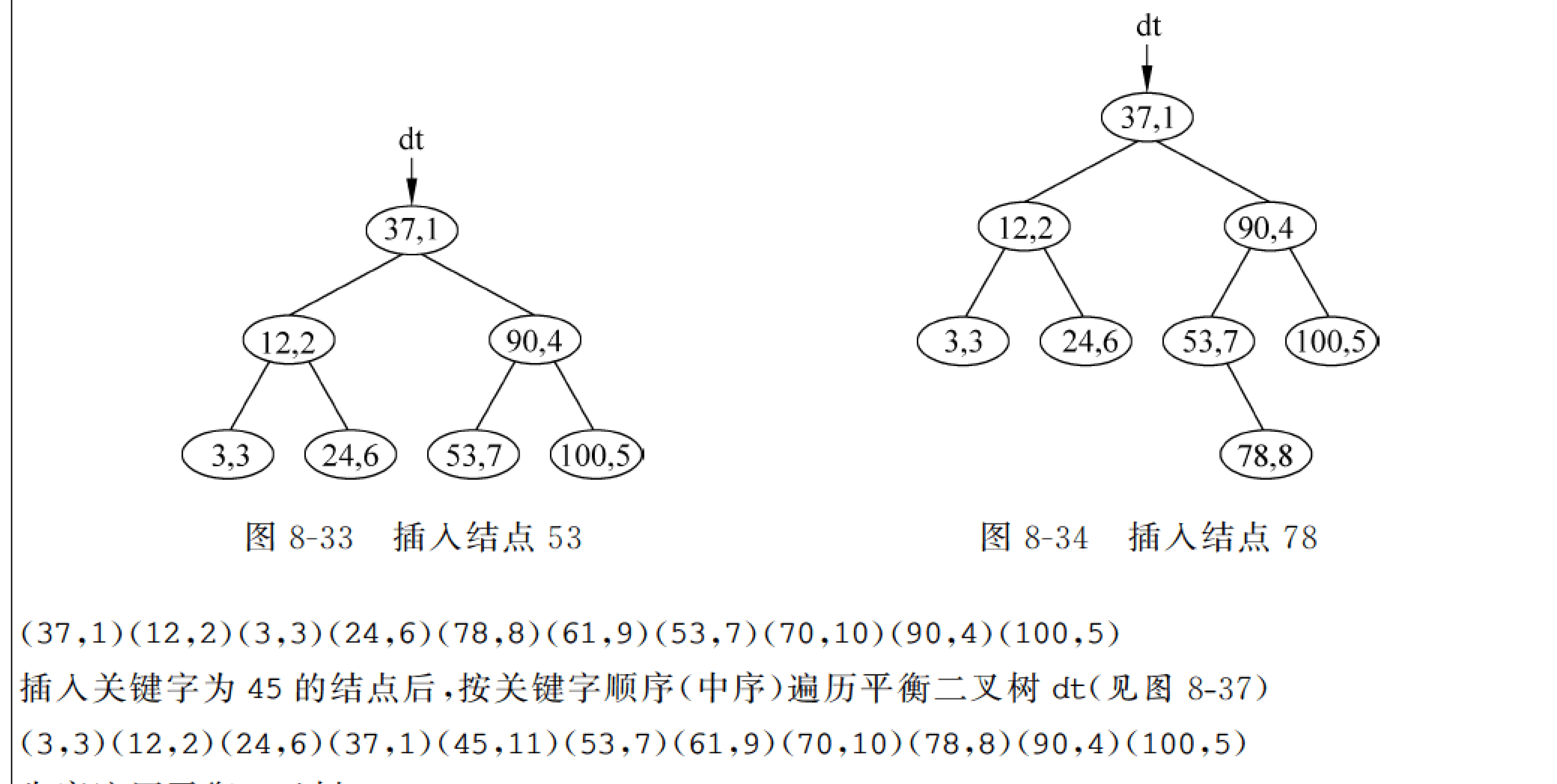



图 8-33 插入结点 53

图 8-34 插入结点 78

(37,1)(12,2)(3,3)(24,6)(78,8)(61,9)(53,7)(70,10)(90,4)(100,5)
插入关键字为 45 的结点后，按关键字顺序(中序)遍历平衡二叉树 dt(见图 8-37)
(3,3)(12,2)(24,6)(37,1)(45,11)(53,7)(61,9)(70,10)(78,8)(90,4)(100,5)
先序遍历平衡二叉树 dt
(61,9)(37,1)(12,2)(3,3)(24,6)(53,7)(45,11)(78,8)(70,10)(90,4)(100,5)
平衡二叉树 dt 中已存在关键字为 53 的数据，故(53,12)无法插入到 dt 中。
请输入待查找的关键字的值：24↙
dt 中存在关键字为 24 的结点，其值为(24,6)。

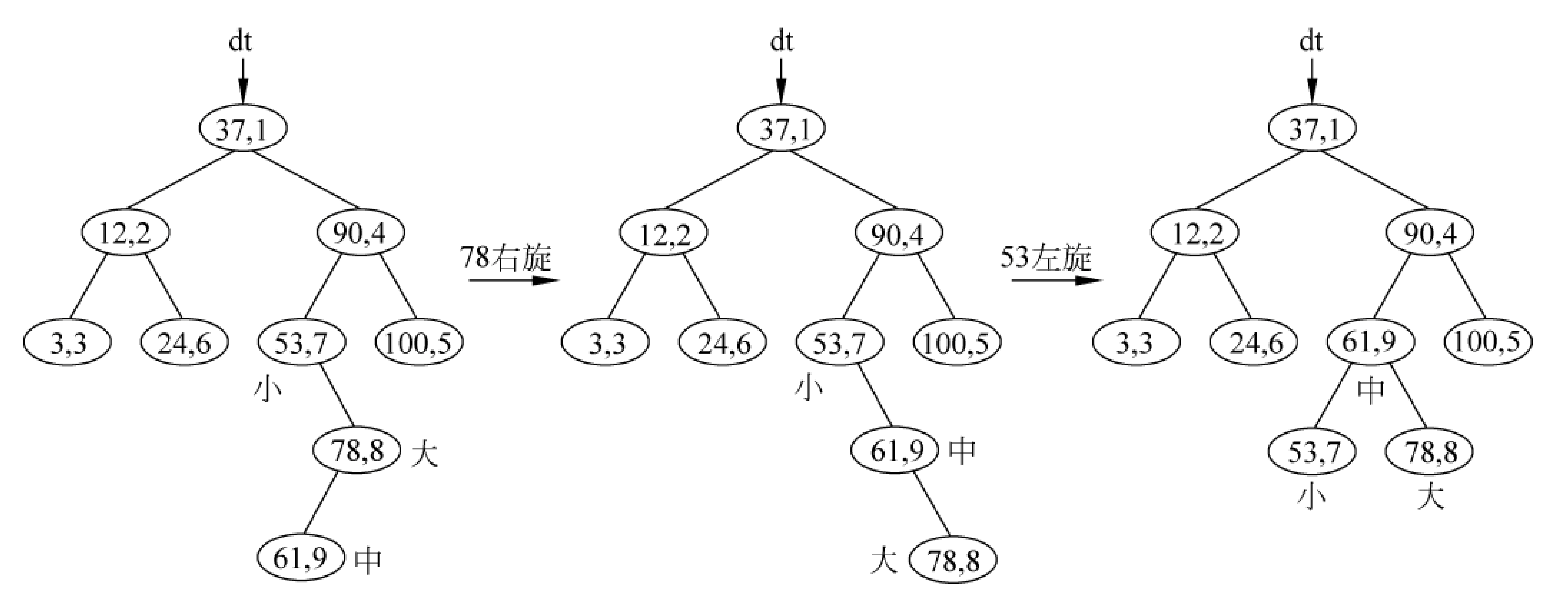



图 8-35 插入结点 61，RL 型不平衡

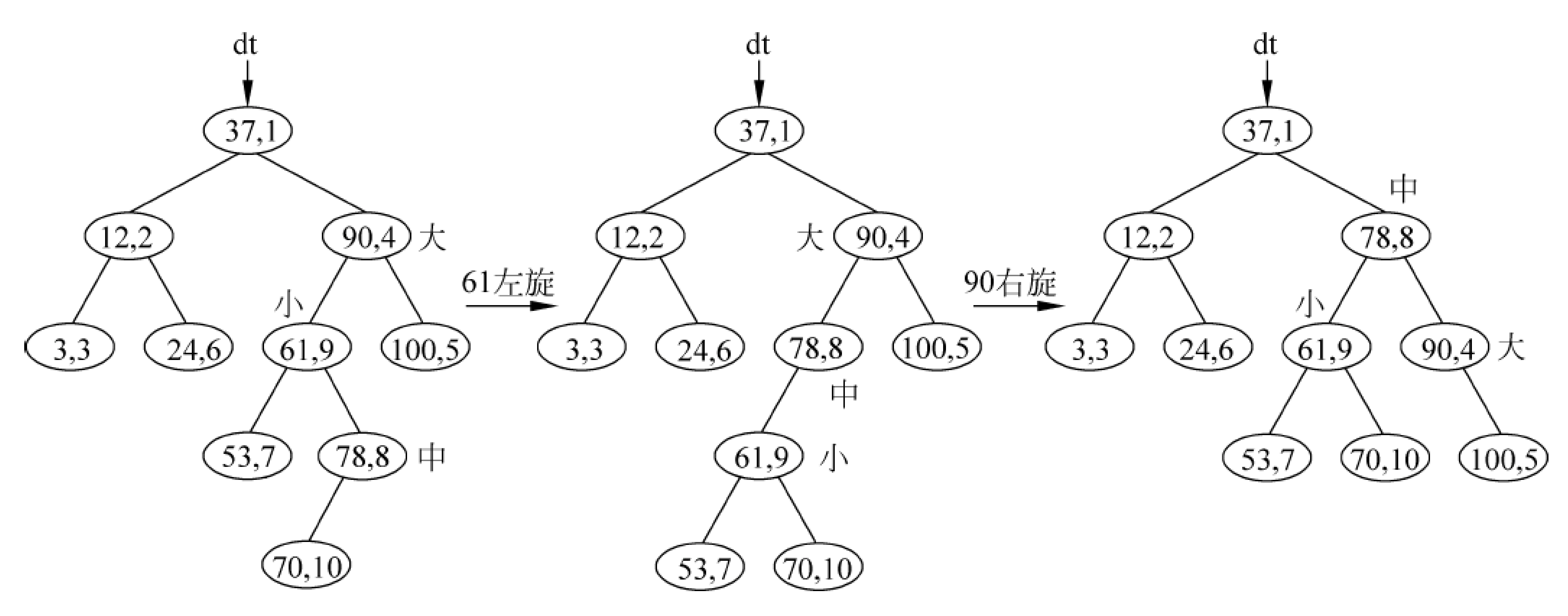



图 8-36 插入结点 70，LR 型不平衡

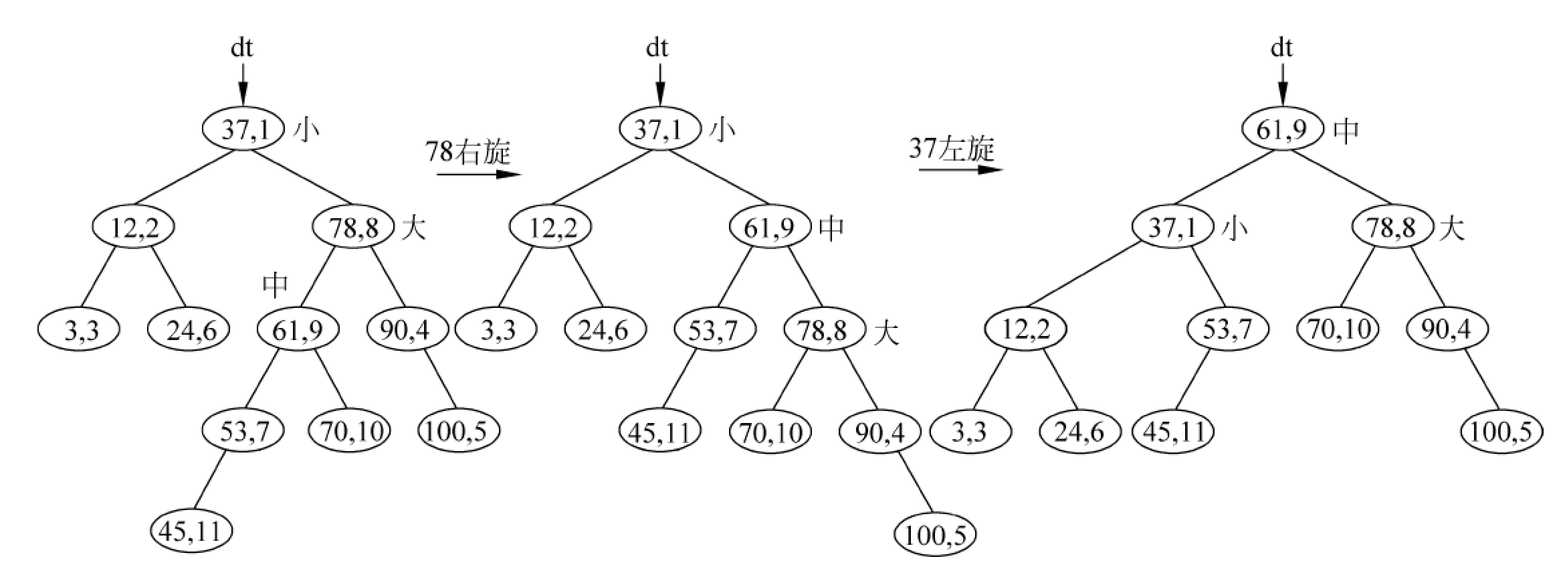



图 8-37　插入结点 45,RL 型不平衡

algo8-5. cpp 和 algo8-4. cpp 在构造二叉树时用的是同一个数据文件,即它们是以相同的顺序插入相同的结点来构造二叉树的。但比较图 8-15 和图 8-37 可见,它们构造的二叉树的形态差别很大,平衡二叉树的深度接近最低,受结点输入顺序的影响不大,而二叉排序树的形态则完全受结点输入顺序的影响。

平衡二叉树插入结点的方法和二叉排序树一样,都是从根结点起,根据待插入结点的关键字是大于、小于还是等于当前结点的关键字,决定在右子树中、在左子树中还是不插入结点。所不同的是,平衡二叉树每插入一个结点就从插入结点起向根结点方向通过考察祖先结点平衡因子的变化来逐层检查是否仍为平衡二叉树。如果由于插入结点导致该结点的某个祖先结点失衡,则要对以这个祖先结点为根的树作平衡旋转处理。方法是通过调换根结点,使左右子树的深度相等。在调换过程中还要保证二叉树的有序性。

因为新插入的结点总是叶子结点,所以它本身总是平衡的(bf=0)。插入结点也不会导致它的双亲结点不平衡(对于它的双亲结点来说,插入结点的这棵子树的深度由 0 变为 1,另一棵子树的深度只能是 0 或 1,结果仍是平衡的)。因此,插入结点如果导致平衡二叉树失衡,则失衡的结点只会是新插入结点的祖先结点(不包括双亲结点)。所以插入结点导致平衡二叉树失衡只有 4 种情况:

(1) 在左子树的左孩子分支上插入结点,导致失衡,称 LL(/)型(从失衡结点起,沿着插入结点方向的连续 3 个结点形如"/"),如图 8-21 所示;

(2) 在左子树的右孩子分支上插入结点,导致失衡,称 LR(＜)型(从失衡结点起,沿着插入结点方向的连续 3 个结点形如"＜"),如图 8-23 所示;

(3) 在右子树的右孩子分支上插入结点,导致失衡,称 RR(\)型(从失衡结点起,沿着插入结点方向的连续 3 个结点形如"\"),如图 8-24 所示;

(4) 在右子树的左孩子分支上插入结点,导致失衡,称 RL(＞)型(从失衡结点起,沿着插入结点方向的连续 3 个结点形如"＞"),如图 8-26 所示。

**注意**:在图 8-21、图 8-23、图 8-24 和图 8-26 平衡旋转前的状态中,只有 T 所指结点的平衡因子是插入结点前的,其他结点的平衡因子都是插入结点后调整过的。因为插入结点后总是由新插结点逐层向根结点调整。上述 4 幅图所描述的正是对树 T 进行的平衡调整。

读者可能会想,除了在图 8-21、图 8-23、图 8-24 和图 8-26 所示的 8 种需进行平衡调整

的状态外，是否还有其他的状态（如各子树的深度与图中所列不同）也需进行平衡调整？答案是否定的。因为新插结点只使得 T 所指结点失衡，而没有使其子孙结点失衡。

（1）、（2）两种情况调用 LeftBalance()函数（算法 9.12）将失衡的二叉树重新调整为平衡二叉树。

对于情况（2），即 LR（<）型失衡，如图 8-22 所示，在 LeftBalance()函数中先对左子树（小值结点）调用 L_Rotate()做左旋，使中值结点上升 1 层，再调用 R_Rotate()做右旋。其新插结点的具体位置有 3 种情况，如图 8-23 所示，分别是：插在中值结点的左（情况①）、右（情况③）子树中，或者新插结点就是中值结点本身（情况②）。新插结点的具体位置虽不影响旋转操作，却影响大、小值 2 结点旋转后的平衡因子，故要区分这 3 种情况来确定它们的平衡因子。

对于情况（1），即 LL（/）型失衡，在 LeftBalance()函数中只调用 R_Rotate()做右旋。如图 8-19 所示，新插结点是在小值结点这棵子树中。由于在右旋过程中，小值结点这棵子树并不被重接，所以无论新插结点是小值结点本身或是插在它的左右子树中都不影响中值或大值结点的平衡因子，所以不需要分情况处理，如图 8-21 所示。

（3）、（4）两种情况分别是（1）、（2）两种情况的镜像，对应地调用函数 RightBalance()，不再赘述。

4 种不平衡情况经调整都有如下特性：

（1）成为 A（∧）型，调整前的中值结点成为新的根结点，其平衡因子为 0，其左、右孩子分别是小值、大值结点；

（2）调整后树的深度和插入结点前一样。

特性（1）解释了对于 LR 型和 RL 型要作两次单旋的原因。以图 8-37 为例，第 1 次以结点 78 为根右旋，虽然未改变子树的深度，但中值结点（61）也就是新根结点由第 3 层调到了第 2 层，再经过以结点 37 为根的左旋，中值结点（61）被调到了第 1 层，成为根结点。

根据特性（1）和平衡二叉树的有序性，很容易对因插入结点失衡的 AVL 树作平衡旋转处理。以图 8-37 为例：插入结点 45 导致结点 37 失衡。由结点 37 沿插入结点方向取直接下两层的结点 78 和 61，构成小、大、中值 3 结点。平衡旋转后的树应以中值结点 61 为根，结点 37 和 78 分别为 61 的左右孩子，形成“∧”型。这样，结点 61 原先的左右孩子结点 53 和 70 失去了双亲结点，而结点 78 和 37 分别失去了左右孩子结点。根据平衡二叉树的有序性，显然应将结点 53 作为 37 的右孩子，结点 70 作为 78 的左孩子。处理结果和调用函数 RightBalance()是一样的。

bo8-3.cpp 中的函数 LR_Rotate()和 RL_Rotate()就是用这种方法直接处理 LR 和 RL 失衡的。启用 algo8-5.cpp 的第 9 行，在函数 LeftBalance()和 RightBalance()中就分别调用了函数 LR_Rotate()和 RL_Rotate()，其运行结果完全一样。

在 LeftBalance()中调用 L_Rotate(T—>lchild)和 R_Rotate(T)处理 LR 型不平衡的好处是直接使用已有函数，减少了函数的个数；而调用 LR_Rotate(T)更能揭示旋转处理的本质，二者各有千秋。

特性（2）说明如果插入一个结点导致平衡二叉树失衡，则只须在通向根结点的路径上进

行一次平衡调整。以图 8-36 为例,插入结点 70 后,导致结点 90 和结点 37 失衡。调整了距新插结点 70 较近的祖先结点 90 为根的子树后,结点 37 的右子树深度和插入结点 70 前一样,也就不用再调整结点 37 了。

平衡二叉树在插入结点后,要向根结点方向逐层考察是否导致某个祖先结点失衡。因为 InsertAVL()是递归函数,在插入结点时是从根结点一直查找到子树为空才插入,使新插结点为叶子结点。递归函数返回时恰是由新插的叶子结点逐层退回到根结点。所以函数 InsertAVL()可以由叶子到根逐层通过返回值向双亲结点传递是否插入了结点的信息。

如果平衡二叉树中已存在待插入结点,则 InsertAVL()不再插入结点,返回值为 FALSE,不作任何处理;如果平衡二叉树中不存在待插入结点,则 InsertAVL()插入结点,返回值为 TRUE。同时还通过引用参数 taller,向双亲结点传递插入结点是否导致子树“长高”的信息。

如果插入结点没有导致子树“长高”,则不会失衡。如图 8-33 所示,插入结点 53 并没有导致其双亲结点 90“长高”,也就不会导致其祖先结点 37“长高”,故不作任何处理。

如果插入结点导致子树“长高”,还要再看是否导致某个祖先结点失衡。如图 8-34 所示,插入结点 78 导致其双亲结点 53“长高”但没有失衡,则不作调整,把“长高”信息继续逐层向根结点方向传递。虽然结点 90 和结点 37“长高”了,但都没有失衡。如图 8-32 所示,插入结点 24 导致其双亲结点 37“长高”但没有失衡,将此“长高”信息向上传递给结点 90,仍然“长高”但没有失衡,继续将“长高”信息向上传递给结点 12,结点 24 的插入导致结点 12 失衡,需对结点 12 作调整。

有些教材定义平衡因子为右子树的深度减去左子树的深度,仅将 LH 定义为－1,RH 定义为＋1 即可。

### 8.2.2 B_树和 $B^+$ 树

B_树是平衡的 m 路查找树,“B”表示平衡。

平衡二叉树的查找效率很高,但在数据量非常大,以至于内存空间不够容纳平衡二叉树所有结点的情况下,就得另辟蹊径。B_树是解决这个问题的一种很好的结构。

```
// c8-4.h 记录类型的结构
struct Record // 记录类型(见图 8-38)
{ KeyType key; // 关键字
  Others others; // 其他部分(由主程定义)
};
```

Record

| key | others |
|---|---|

图 8-38 记录类型

```
// c8-5.h B_树的结点类型。在教科书第 239 页
typedef struct BTNode
{ int keynum; // 结点中关键字个数,即结点的大小
  BTNode * parent; // 指向双亲结点
  KeyType key[m + 1]; // 关键字向量,0 号单元未用
  Record * recptr[m + 1]; // 记录指针向量,0 号单元未用
```

```
  BTNode * ptr[m + 1]; // 子树指针向量
}BTNode, * BTree; // B_树结点和 B_树的类型(见图 8-39)
```

```
// c8-6.h B_树的查找结果类型。在教科书第 240 页
struct Result // B_树的查找结果类型(见图 8-40)
{ BTree pt; // 指向关键字所在的结点
  int i; // 1..m,在结点中的关键字序号
  int tag; // 1:查找成功,0:查找失败
};
```

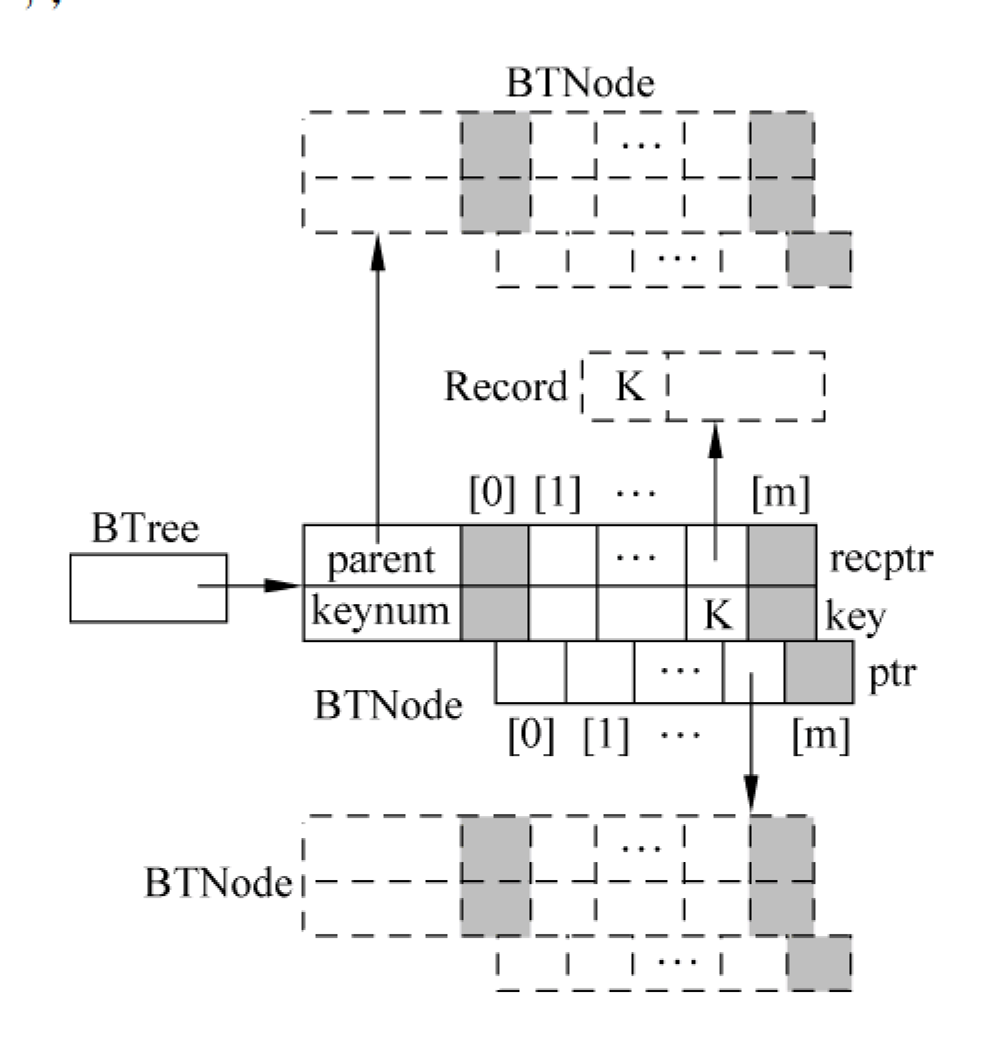




图 8-39 m 阶 B_树结点及指针结构

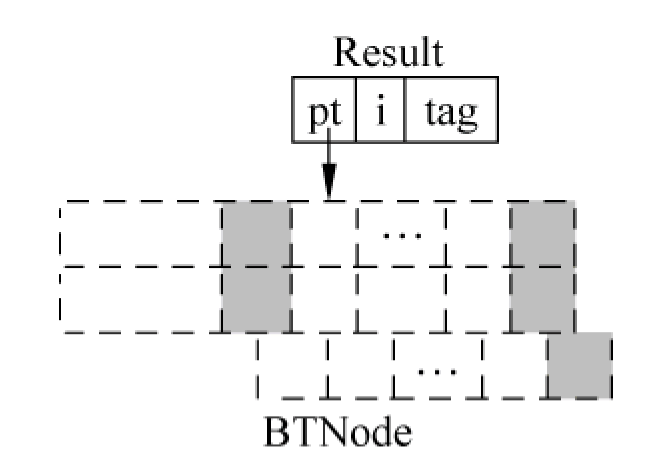



图 8-40 B_树的查找结果类型

B_树的结点类型和前面介绍过的所有静态、动态查找表的结点类型有一个重要区别:它不是把整个记录(或数据)都存放在结点中,而是仅在结点中存放记录的关键字和记录的地址两项。在结点中查找到关键字后,再根据其地址找到记录。当记录所占的存储空间非常大(如人口普查资料,除了身份证号码作为关键字外,还有姓名、民族、籍贯、文化程度、婚姻状况、职业、户口所在地、住址等一系列信息)时,这样做的好处是减小了结点占用的存储空间,也就减小了整个 B_树占用的存储空间。

B_树是 m 路查找树,它的每个结点最多可以有 m−1 个关键字,m 棵子树(m=2 时即为二叉树,有 1 个关键字,2 棵子树)。子树总比关键字的数量多 1,故 key[0]和 recptr[0]单元不用,而 ptr[0]单元要用。[m]单元在正常情况下是不用的,只是在结点的 keynum=m−1,又向该结点插入关键字时,临时占用[m]单元。然后就要把该结点尽量平均地分裂成 2 个结点。

```
// bo8-4.cpp B_树的基本操作,包括算法 9.13 和算法 9.14
void InitDSTable(BTree &DT)
{ // 操作结果:构造一个空的 B_树 DT(见图 8-41)
  DT = NULL;
}
void DestroyDSTable(BTree &DT)
{ // 初始条件:B_树 DT 存在。操作结果:销毁 DT
```

DT

| NULL |
|---|

图 8-41 空的 B_树 DT

```
    int i;
    if(DT) // 非空树
    { for(i = 0;i<= DT->keynum;i++ ) // 由第 0 棵到第 keynum 棵
        DestroyDSTable(DT->ptr[i]); // 依次递归销毁第 i 棵子树
      free(DT); // 释放根结点
      DT = NULL; // 空指针赋 0
    }
}
void TraverseDSTable(BTree DT,void(* Visit)(BTNode,int))
{ // 初始条件:B_树 DT 存在,Visit 是对结点操作的应用函数
  // 操作结果:按关键字的顺序对 DT 的每个结点调用函数 Visit()一次且至多一次
    int i;
    if(DT) // 非空树
      for(i = 0;i<= DT->keynum;i++ ) // 对于 DT 的所有关键字和子树
      { if(i>0) // 有关键字
          Visit( * DT,i); // 访问 DT 的第 i 个关键字及记录指针
        if(DT->ptr[i]) // DT 有第 i 棵子树
          TraverseDSTable(DT->ptr[i],Visit);
          // 对 DT 的第 i 棵子树,递归调用 TraverseDSTable()
      }
}
int Search(BTNode* p,KeyType K)
{ // 采用折半查找法在 p->key[1..keynum]中查找 i,使得 p->key[i]≤K<p->key[i+1]
    int mid,low = 1,high = p->keynum; // 置区间初值
    if LT(K,p->key[low]) // 关键字太小,p->key[1..keynum]中不存在 K
      return 0; // 返回 0
    while(low<high) // 当查找范围大于 1
    { mid = (low + high + 1)/2; // 中值(取偏大)
      if EQ(K,p->key[mid]) // 中值是 K
        return mid; // 返回其序号
      if LT(K,p->key[mid]) // K 小于中值
        high = mid - 1; // 继续在前半区间进行查找
      else // K 大于中值
        low = mid; // 继续在包括[mid]的后半区间进行查找
    }
    return low; // low = high,查找范围等于 1 且 p->key[1..keynum]中不存在 K
}
Result SearchBTree(BTree T,KeyType K)
{ // 在 m 阶 B_树 T 上查找关键字 K,返回结果(pt,i,tag)。若查找成功,则特征值
  // tag = 1,指针 pt 所指结点中第 i 个关键字等于 K;否则特征值 tag = 0,等于 K 的
  // 关键字应插入在指针 Pt 所指结点中第 i 和第 i+1 个关键字之间。算法 9.13
    BTree p = T,q = NULL; // 初始化,p 指向待查结点,q 指向 p 的双亲
    Status found = FALSE; // 查找成功标志,初值为未成功
    int i = 0;
    Result r; // 查找结果存于此变量,以便返回
```

```
    while(p&&!found) // 待查结点 * p 不为空且未找到 K
    { i = Search(p,K); // 在结点 * p 中查找关键字 K
      if(i>0&&p->key[i] == K) // 找到
        found = TRUE; // 置查找成功标志为真
      else // 未找到
      { q = p; // 继续向下找,当前结点成为新的双亲结点
        p = p->ptr[i]; // p 指向继续查找的结点
      }
    }
    if(found) // 查找成功
    { r.pt = p; // r.pt 指向关键字 K 所处的结点
      r.tag = 1; // 查找成功标志
    }
    else // 查找不成功,返回 K 的插入位置信息
    { r.pt = q; // r.pt 指向关键字 K 应插入的结点
      r.tag = 0; // 查找不成功标志
    }
    r.i = i; // r.i 指示 r.pt 所指结点中关键字 K 所在(找到)或应在其后插入(未找到)的序号
    return r; // 返回结果(pt,i,tag)
  }
  void split(BTree q,BTree &ap)
  { // 将结点 * q 分裂成两个结点,前一半保留在 * q,后一半移入新生结点 * ap(见图 8-42)
```

(a) 分裂前　　(b) 分裂后

图 8-42　调用 split()示例(m=3,分裂后,(r2,k2)插到 * q 的双亲结点中,q 和 ap 作为它的左右子树)

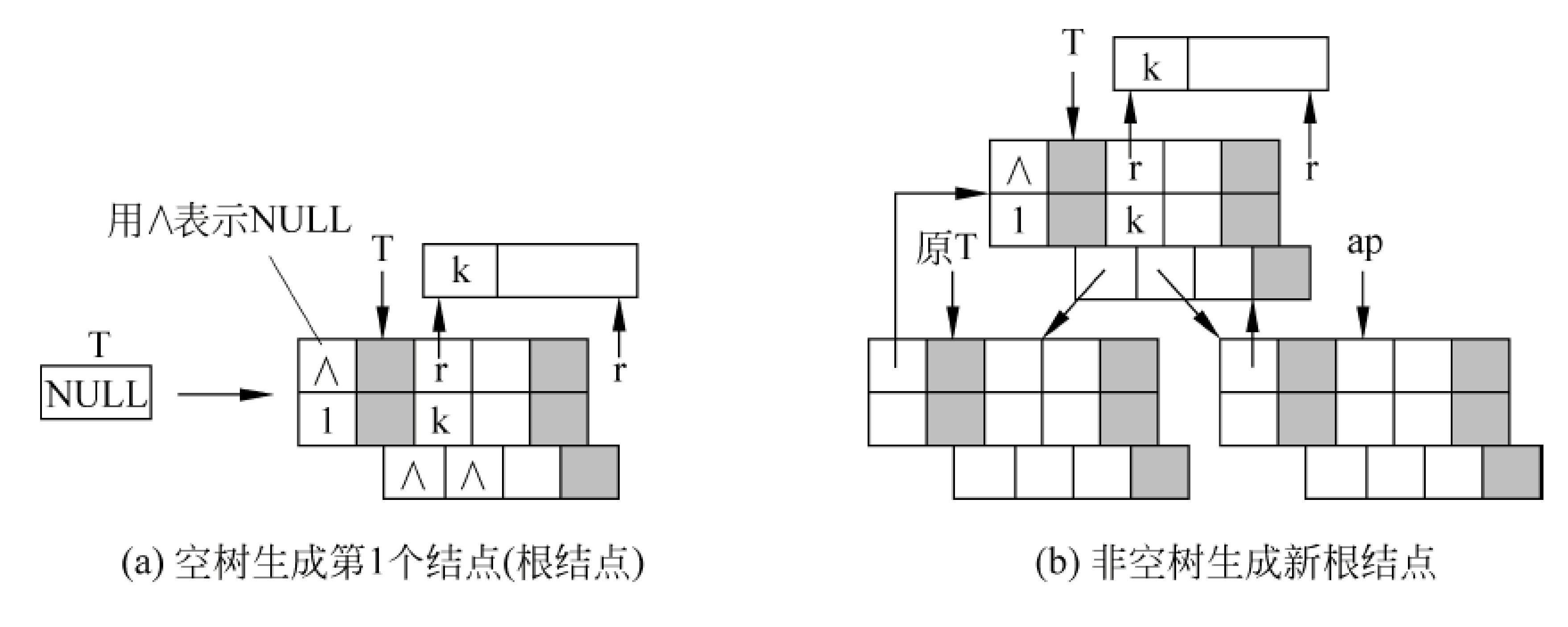
```
    int i;
    ap = (BTree)malloc(sizeof(BTNode)); // 生成新结点 * ap
    ap->ptr[0] = q->ptr[s]; // 结点 * q 的后一半移入结点 * ap
    if(ap->ptr[0]) // ap->ptr[0]不为空
      ap->ptr[0]->parent = ap; // 给 ap->ptr[0]的双亲域赋值 ap
    for(i = s + 1;i<= m;i++ ) // 对于 * q 中后一半数据
    { ap->key[i - s] = q->key[i]; // 3 个成员均移结点 * ap
      ap->recptr[i - s] = q->recptr[i];
      ap->ptr[i - s] = q->ptr[i];
      if(ap->ptr[i - s]) // ap->ptr[i - s]不为空
        ap->ptr[i - s]->parent = ap; // 给 ap->ptr[i - s]的双亲域赋值 ap
    }
    ap->keynum = m - s; // 新结点 * ap 的关键字个数
    q->keynum = s - 1; //  * q 的前一半保留,修改 * q 的关键字个数
  }
  void Insert(BTree q,int i,Record* r,BTree ap) // (见图 8-43)
```

```
{ // 将记录地址 r 和 r->key 分别赋给 q->recptr[i+1]和 q->key[i+1],q->ptr[i+1]指向结点 * ap
  int j;
  for(j = q->keynum;j>i;j-- ) // 由后到前,空出( * q)[i+1]
  { q->key[j+1] = q->key[j]; // 3 个成员均向后移
    q->recptr[j+1] = q->recptr[j];
    q->ptr[j+1] = q->ptr[j];
  }
  q->key[i+1] = r->key; // 将 r->key 赋给 q->key[i+1]
  q->recptr[i+1] = r; // 将记录地址 r 赋给 q->recptr[i+1]
  q->ptr[i+1] = ap; // q->ptr[i+1]指向结点 * ap(ap 是 r 的右孩子)
  if(ap) // ap 不空
    ap->parent = q; // q 是 ap 的双亲所在的结点
  q->keynum++; // 结点 * q 的关键字数量加 1
}
```

(a) 插入前　(b) 插入后

图 8-43　调用 Insert()示例(i=1)

```
void NewRoot(BTree &T,Record* r,BTree ap)
{ // 生成含信息(T,r,ap)的新的根结点 * T,原根结点 T 和 ap 为其子树指针(见图 8-44)
```




(a) 空树生成第1个结点(根结点)　(b) 非空树生成新根结点

图 8-44　调用 NewRoot()示例

```
  BTree p = (BTree)malloc(sizeof(BTNode)); // 动态生成新根结点
  p->parent = NULL; // 新根结点的双亲为空
  p->keynum = 1; // 新根结点有 1 个关键字
  p->key[1] = r->key; // 这个关键字是记录 r 的关键字
  p->recptr[1] = r; // 指向记录 r
  p->ptr[0] = T; // 原根结点 T 为新根结点的第 1 棵子树
  if(T) // 原根结点 T 不空
    T->parent = p; // 新根结点是原根结点 T 的双亲
  p->ptr[1] = ap; // 结点 * ap 为新根结点的第 2 棵子树
  if(ap) // ap 不空
    ap->parent = p; // 新根结点是 ap 的双亲
  T = p; // T 指向新根结点
}
void InsertBTree(BTree &T,Record* r,BTree q,int i)
{ // 在 m 阶 B_树 T 上结点 * q 的 key[i]与 key[i+1]之间插入关键字 r->k 和地址 r。若引起
  // 结点过大,则沿双亲链进行必要的结点分裂调整,使 T 仍是 m 阶 B_树。修改算法 9.14
  BTree ap = NULL; // 空结点
```

```
  Status finished = FALSE; // 插入完成标志,初始为未完成
  while(q&&!finished) // q 不空且未完成插入
  { Insert(q,i,r,ap); // 将 r->key 和记录地址 r 分别赋给 q->key[i+1]和 q->recptr[i+1],
                      // q->ptr[i+1]指向结点 * ap
    if(q->keynum<m) // 关键字未超出结点的容量
      finished = TRUE; // 插入完成
    else // 关键字超出了结点的容量,分裂结点 * q
    { r = q->recptr[s]; // 分裂点的记录地址赋给 r
      split(q,ap); // 将 q->key[s+1..m],q->recptr[s+1..m]和 q->ptr[s..m]移入结点 * ap
      // 结点 * q 中仅保留 q->key[1..s-1],q->recptr[1..s-1]和 q->ptr[0..s-1]
      q = q->parent; // 当前结点为被分裂结点的双亲结点
      if(q) // 被分裂结点的双亲结点存在
        i = Search(q,r->key); // 在被分裂结点的双亲结点 * q 中查找 r->key 的插入位置
    }
  }
  if(!finished) // T 是空树(参数 q 初值为 NULL)或根结点已分裂为结点 * q 和 * ap
    NewRoot(T,r,ap); // 生成含信息(T,r,ap)的新的根结点 * T,原 T 和 ap 为子树指针
}
```

```
// func8-6.cpp 包括对 B_树的输入输出操作
void Visit(BTNode c,int i) // TraverseDSTable()调用的与之配套的访问记录的函数
{ printf("(%d,%d)",c.recptr[i]->key,c.recptr[i]->others.order);
}
void InputKey(KeyType &k) // 与之配套的由键盘输入关键字的函数
{ scanf("%d",&k);
}
```

```
// algo8-6.cpp 检验 bo8-4.cpp 的程序
#include"c1.h"
#define m 3 // B_树的阶,现设为 3
int s = (m+1)/2; // s 为分裂结点的中值
typedef int KeyType; // 设关键字域为整型
struct Others // 记录的其他部分
{ int order; // 整型变量,顺序
};
#include"c8-2.h" // 对两个数值型关键字比较的约定
#define N 12 // 记录数组的数据元素个数
#include"c8-4.h" // 记录类型
#include"c8-5.h" // B_树的结点类型
#include"c8-6.h" // B_树的查找结果类型
#include"bo8-4.cpp" // B_树的基本操作,包括算法 9.13 和算法 9.14
#include"func8-6.cpp" // 包括对 B_树的输入输出操作
void main()
{
  Record r[N] = {{24,1},{45,2},{53,3},{12,4},{37,5},{50,6},{61,7},{90,8},
  {100,9},{70,10},{3,11},{37,12}}; // (记录元素存于数组中,以教科书中图 9.16(a)为例)
```

```
    BTree t;
    Result u;
    KeyType j;
    int i;
    InitDSTable(t); // 构造空的 B_树 t
    for(i = 0;i<N;i++ ) // 将记录数组 r[N]的数据依次插入树 t 中
    { u = SearchBTree(t,r[i].key); // 在树 t 中查找是否已存在关键字为 r[i].key 的记录
      if(u.tag) // 在树 t 中已存在关键字为 r[i].key 的记录
        printf("树 t 中已存在关键字为 %d 的记录,故(%d, %d)无法插入。\n",r[i].key,
        r[i].key,r[i].others.order);
      else // 在树 t 中不存在关键字为 r[i].key 的记录
        InsertBTree(t,&r[i],u.pt,u.i);
        // 将 r[i]的关键字和地址插入到 t 中结点 u.pt 的[u.i]和[u.i + 1]之间
    }
    printf("按关键字的顺序遍历 B_树 t:\n");
    TraverseDSTable(t,Visit); // 按关键字的顺序遍历 B_树 t
    for(i = 1;i<= 4;i++ ) // 4 次在树 t 中查找给定关键字的数据
    { printf("\n 请输入待查找记录的关键字:");
      InputKey(j); // 输入关键字 j
      u = SearchBTree(t,j); // 在树 t 中查找关键字为 j 的数据
      if(u.tag) // 找到
        Visit( * (u.pt),u.i); // 输出查找到的记录
      else // 未找到
        printf("未找到");
    }
    printf("\n");
    DestroyDSTable(t); // 销毁树 t
}
```

**程序运行结果**(见图 8-45):

```
树 t 中已存在关键字为 37 的记录,故(37,12)无法插入。
按关键字的顺序遍历 B_树 t:
(3,11)(12,4)(24,1)(37,5)(45,2)(50,6)(53,3)(61,7)(70,10)(90,8)(100,9)
请输入待查找记录的关键字:90↙
(90,8)
请输入待查找记录的关键字:37↙
(37,5)
请输入待查找记录的关键字:40↙
未找到
请输入待查找记录的关键字:55↙
未找到
```

函数 SearchBTree()返回在 B_树中查找关键字的结果。如找到,结果指示关键字所在的结点及在该结点中的序号,如图 8-45 中的 u(1)和 u(2);如未找到,结果指示关键字应插入的结点及其前驱关键字在结点中的序号,如图 8-45 中的 u(3)和 u(4)。注意:其前驱关

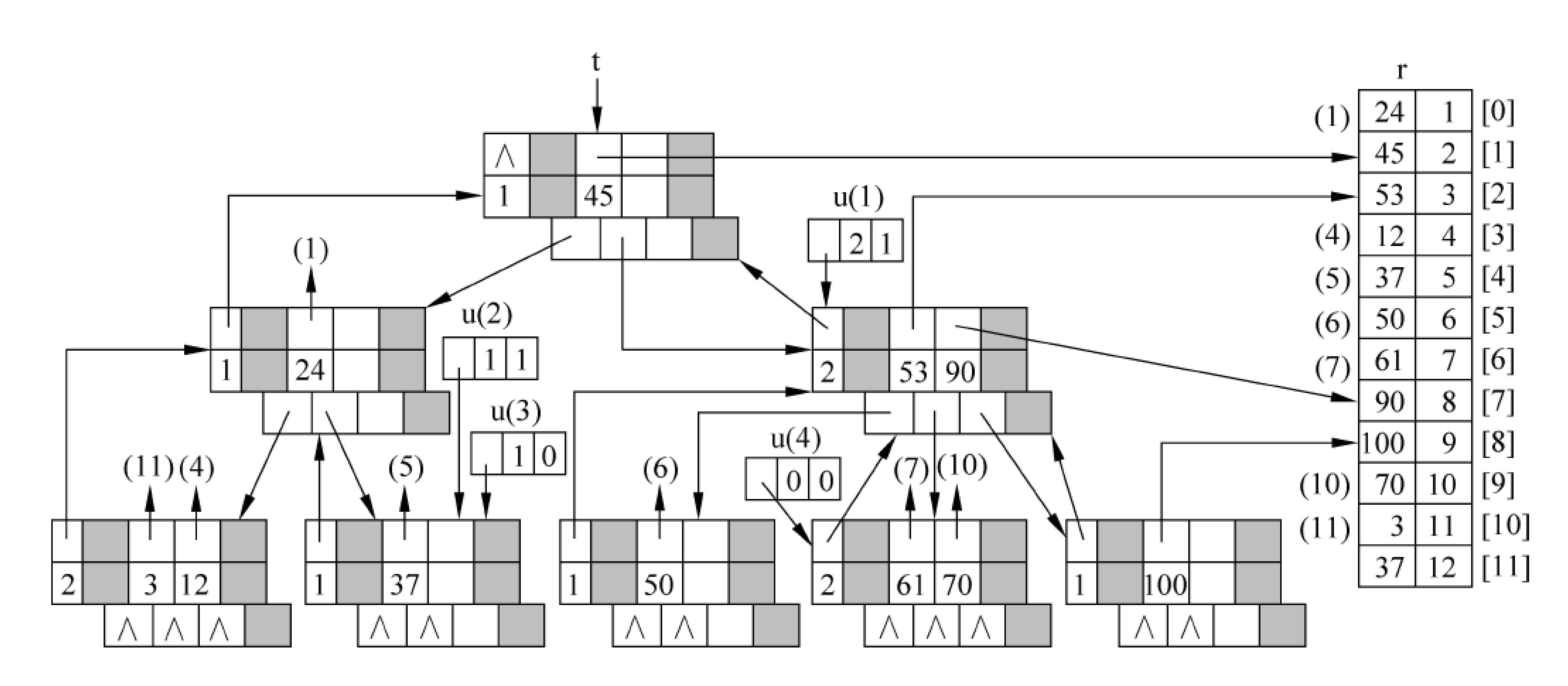



图 8-45　algo8-6.cpp 程序运行结果

键字的右孩子(或其后继关键字的左孩子)一定是空。

algo8-6.cpp 调用 bo8-4.cpp 中的算法 9.13 仅对 B_树作了示意性的描述。B_树是用于处理大数据量的查找操作的。其中的记录量大到不能够存放在内存数组 r 中,要将记录存放在外存的多个文件中。甚至内存中也放不下整个 B_树,内存中只能存放 B_树的 1 个结点。所以,B_树的每个结点都存于外存的文件中。查找过程是首先将 B_树的根结点文件放入内存中,依据关键字进行查找。随时关闭查找过的 B_树结点,再在内存中随时打开新的 B_树结点,直至查找结束。为了提高查找速度,就要尽量减少打开、关闭文件的次数。则要尽量增大 B_树每个结点可容纳的关键字数。

## 8.2.3　键树

键树用于关键字为字符串的情况,故键树也称为“词典查找树”。

```
// c8-7.h 键树记录的存储结构。在教科书第 248 页
#define MAX_KEY_LEN 16 // 关键字串的最大长度
struct KeyType // 关键字串类型,类似串的定长顺序存储结构(见图 8-46)
{ char ch[MAX_KEY_LEN]; // 关键字串
  int num; // 关键字串长度
};
enum NodeKind{LEAF,BRANCH}; // 结点种类:{叶子,分支}
```

KeyType
ch
…　num
[0]　[ ]
MAX_KEY_LEN−1

图 8-46　KeyType 关键字串类型

```
// c8-8.h 双链键树的存储结构。在教科书第 248 页
typedef struct DLTNode // 双链树类型(见图 8-47)
{ char symbol; // 关键字符
  DLTNode *next; // 指向右兄弟结点的指针
  NodeKind kind; // 结点种类
  union // 共用体
  { Record *infoptr; // 叶子结点的记录指针
    DLTNode *first; // 分支结点的孩子链指针
```

DLTree DLTNode(分支结点)
symbol next BRANCH first
DLTNode DLTNode

DLTree DLTNode(叶子结点)
symbol next LEAF infoptr
key others
DLTNode Record

图 8-47 双链树存储结构

```
  };
}DLTNode, * DLTree;
```

```
// bo8-5.cpp 双链键树的基本操作,包括算法 9.15
void InitDSTable(DLTree &DT)
{ // 操作结果：构造一个空的双链键树 DT(见图 8-48)
  DT = NULL;
}
void DestroyDSTable(DLTree &DT)
{ // 初始条件：双链键树 DT 存在。操作结果：销毁 DT
  if(DT) // 非空树
  { if(DT->kind == BRANCH) // * DT 是分支结点
      DestroyDSTable(DT->first); // 递归销毁孩子子树
    DestroyDSTable(DT->next); // 递归销毁兄弟子树
    free(DT); // 释放根结点
    DT = NULL; // 空指针赋 0
  }
}
```

DT
NULL

图 8-48 空的双链键树 DT

```
Record* SearchDLTree(DLTree T,KeyType K)
{ // 在双链键树 T 中查找关键字串等于 K 的记录,若存在,
  // 则返回指向该记录的指针；否则返回空指针。修改算法 9.15
  DLTree p = T; // 初始化
  int i = 0;
  if(T) // 树不空
  { while(p&&i<K.num) // * p 不空且未到最后一个字符
    { while(p&&p->symbol != K.ch[i]) // 查找关键字的第 i 位
        p = p->next; // 顺序在右兄弟结点中查找
      if(p&&i<K.num) // 准备查找下一位
        p = p->first; // p 指向孩子结点
      ++i; // 关键字向后移一位
    } // 查找结束
    if(!p) // 查找不成功
      return NULL;
    else // 查找成功
```

```
        return p->infoptr; // 返回指向该记录的指针
    }
    else // 树空
        return NULL;
}
void InsertDSTable(DLTree &DT,Record* r)
{ // 初始条件：双链键树 DT 存在，r 为待插入的数据元素的指针
  // 操作结果：若 DT 中不存在其关键字串等于(*r).key.ch 的数据元素，则按关键字顺序插 r 到 DT 中
  DLTree p = NULL,q,ap;
  int i = 0;
  KeyType K = r->key;
  if(!DT&&K.num) // 空树且关键字符串非空
  { DT = ap = (DLTree)malloc(sizeof(DLTNode)); // 动态生成根结点
    for(;i<K.num;i++) // 插入分支结点
    { if(p) // p 不空(不是第一次生成结点)
        p->first = ap; // p 的孩子指针指向新生成的结点
      ap->next = NULL; // 右兄弟为空
      ap->symbol = K.ch[i]; // 关键字符为当前位关键字
      ap->kind = BRANCH; // 结点种类为分支
      p = ap; // p 指向 ap
      ap = (DLTree)malloc(sizeof(DLTNode)); // 动态生成孩子结点
    }
    p->first = ap; // 插入叶子结点(见图 8-49)
    ap->next = NULL; // 叶子结点的右兄弟为空
    ap->symbol = Nil; // 叶子结点的关键字符为空
    ap->kind = LEAF; // 结点种类为叶子
    ap->infoptr = r; // 记录指针指向关键字串所在记录
  }
  else // 非空树
  { p = DT; // 指向根结点
    while(p&&i<K.num) // *p 不空且 i 小于关键字串的长度
    { while(p&&p->symbol<K.ch[i]) // 沿右兄弟结点查找当前关键字符
      { q = p; // q 指向当前结点
        p = p->next; // p 指向当前结点的右兄弟结点
      }
      if(p&&p->symbol == K.ch[i]) // 找到与 K.ch[i]相符的结点
      { q = p; // q 指向当前结点
        p = p->first; // p 指向将与 K.ch[i+1]比较的结点(孩子结点)
        ++i; // 关键字向后移一位
      }
      else // 未找到与 K.ch[i]相符的结点，插入关键字
      { ap = (DLTree)malloc(sizeof(DLTNode)); // 动态生成结点
        if(q->first == p)
          q->first = ap; // 在长子的位置插入
```



图 8-49　叶子结点

```
            else // q->next == p
              q->next = ap; // 在右兄弟的位置插入
            ap->next = p; // 右兄弟为 p
            ap->symbol = K.ch[i]; // 关键字符为当前位关键字
            ap->kind = BRANCH; // 结点种类为分支
            p = ap; // p 指向 ap
            ap = (DLTree)malloc(sizeof(DLTNode)); // 动态生成结点
            i++; // 关键字向后移一位
            for(;i<K.num;i++) // 插入分支结点
            { p->first = ap; // 在长子的位置插入
              ap->next = NULL; // 右兄弟为空
              ap->symbol = K.ch[i]; // 关键字符为当前位关键字
              ap->kind = BRANCH; // 结点种类为分支
              p = ap; // p 指向 ap
              ap = (DLTree)malloc(sizeof(DLTNode)); // 动态生成孩子结点
            }
            p->first = ap; // 插入叶子结点(见图 8-49)
            ap->next = NULL; // 叶子结点的右兄弟为空
            ap->symbol = Nil; // 叶子结点的关键字符为空
            ap->kind = LEAF; // 结点种类为叶子
            ap->infoptr = r; // 记录指针指向关键字串所在记录
          }
        }
      }
    }
    struct SElemType // 定义栈元素类型
    { char ch;
      DLTree p;
    };
    #include"c3-1.h" // 顺序栈
    #include"bo3-1.cpp" // 顺序栈的基本操作
    void TraverseDSTable(DLTree DT,void(*Visit)(Record*))
    { // 初始条件：双链键树 DT 存在，Visit 是对记录操作的应用函数
      // 操作结果：按关键字的顺序输出关键字及其对应的记录
      SqStack s;
      SElemType e;
      DLTree p;
      int i = 0,n = 9; // 输出 n 个元素后换行
      if(DT) // 树非空
      { InitStack(s); // 初始化栈
        e.p = DT; // 将根结点的信息赋给 e
        e.ch = DT->symbol;
        Push(s,e); // 将 e 入栈
        p = DT->first; // p 指向根结点的长子
```

```
    while(p->kind == BRANCH) // p是分支结点
    { e.p = p; // 将结点 p 的信息赋给 e
      e.ch = p->symbol;
      Push(s,e); // 将 e 入栈
      p = p->first; // p 指向 p 的长子
    }
    e.p = p; // 将结点 p 的信息赋给 e
    e.ch = p->symbol;
    Push(s,e); // 将 e 入栈
    Visit(p->infoptr); // 访问叶子结点的记录
    i++; // 访问数 + 1
    while(!StackEmpty(s)) // 栈不空
    { Pop(s,e); // 弹出栈顶元素
      p = e.p; // 栈元素指针赋 p
      if(p->next) // p 有右兄弟结点
      { p = p->next; // p 指向 p 的右兄弟
        while(p->kind == BRANCH) // p 是分支结点
        { e.p = p; // 将结点 p 的信息赋给 e
          e.ch = p->symbol;
          Push(s,e); // 将 e 入栈
          p = p->first; // p 指向 p 的长子
        }
        e.p = p; // 将结点 p 的信息赋给 e
        e.ch = p->symbol;
        Push(s,e); // 将 e 入栈
        Visit(p->infoptr); // 访问叶子结点的记录
        i++; // 访问数 + 1
        if(i%n == 0)
          printf("\n"); // 输出 n 个元素后换行
      }
    }
  }
}
```

```
// func8-7.cpp 包括对键树的输入输出操作
void Visit(Record*r) // TraverseDSTable()调用的与之配套的访问记录的函数
{ printf("(%s,%d)",r->key.ch,r->others.order);
}
void InputKey(char*k) // 与之配套的由键盘输入关键字符串的函数
{ scanf("%s",k); // 输入待查找记录的关键字符串给 k
}
```

f8-5.txt 内容如下：

```
CAI 1
CAO 2
LI 3
LAN 4
CHA 5
CHANG 6
WEN 7
CHAO 8
YUN 9
YANG 10
LONG 11
WANG 12
ZHAO 13
LIU 14
WU 15
CHEN 16
LI 17
```

```
// algo8-7.cpp 检验 bo8-5.cpp 的程序
#include"c1.h"
#define N 20 // 数组可容纳的数据元素个数
struct Others // 记录的其他部分
{ int order; // 整型变量,顺序
};
#define Nil '$' // 定义结束符为 $
#include"c8-7.h" // 键树记录的存储结构
#include"c8-4.h" // 记录类型
#include"c8-8.h" // 双链键树的存储结构
#include"bo8-5.cpp" // 双链键树的基本操作
#include"func8-7.cpp" // 包括对键树的输入输出操作
void main()
{
  DLTree t;
  int i,j=0; // 数据个数,初始为 0
  KeyType k;
  Record *p,r[N]; // 记录数组
  FILE *f; // 文件指针类型
  InitDSTable(t); // 构造空的双链键树 t
  f=fopen("f8-5.txt","r"); // 打开数据文件 f8-5.txt
  do // 将数据文件中的记录依次读入 r 并插入树 t
  { i=fscanf(f,"%s%d",&r[j].key.ch,&r[j].others.order); // 由文件输入数据给 r[j]
    if(i!=-1) // 输入数据成功
    { r[j].key.num=strlen(r[j].key.ch); // 给 r[j].key.num 域赋值
      p=SearchDLTree(t,r[j].key); // 在双链键树 t 中查找关键字串等于 r[j].key 的记录
      if(!p) // t 中不存在关键字为 r[j].key 的项
      { InsertDSTable(t,&r[j]); // 将 r[j]的地址及关键字串插入 t 中
```

```
      j++; // 记录数组的数据个数+1
    }
    else // 在树t中已存在关键字为r[j].key的项
      printf("树t中已存在关键字为%s的记录,故(%s,%d)无法插入。\n",
      r[j].key.ch,r[j].key.ch,r[j].others.order);
  }
}while(!feof(f)&&j<N); // 未到数据文件的结尾且记录数组未满
fclose(f); // 关闭数据文件
printf("按关键字符串的顺序遍历树t:\n");
TraverseDSTable(t,Visit); // 遍历双链键树t
printf("\n请输入待查找记录的关键字符串:");
InputKey(k.ch); // 输入待查找记录的关键字符串给k.ch
k.num=strlen(k.ch); // 求得关键字串的长度赋给k.num
p=SearchDLTree(t,k); // 在双链键树t中查找关键字串等于k的记录
if(p) // 存在该记录
  Visit(p); // 输出该记录
else // 不存在该记录
  printf("未找到"); // 输出未找到信息
printf("\n");
DestroyDSTable(t); // 销毁双链键树t
}
```

**程序运行结果**(见图8-50):

```
树t中已存在关键字为LI的记录,故(LI,17)无法插入。
按关键字符串的顺序遍历树t:
(CAI,1)(CAO,2)(CHA,5)(CHANG,6)(CHAO,8)(CHEN,16)(LAN,4)(LI,3)(LIU,14)
(LONG,11)(WANG,12)(WEN,7)(WU,15)(YANG,10)(YUN,9)(ZHAO,13)
请输入待查找记录的关键字符串:LI↙
(LI,3)
```

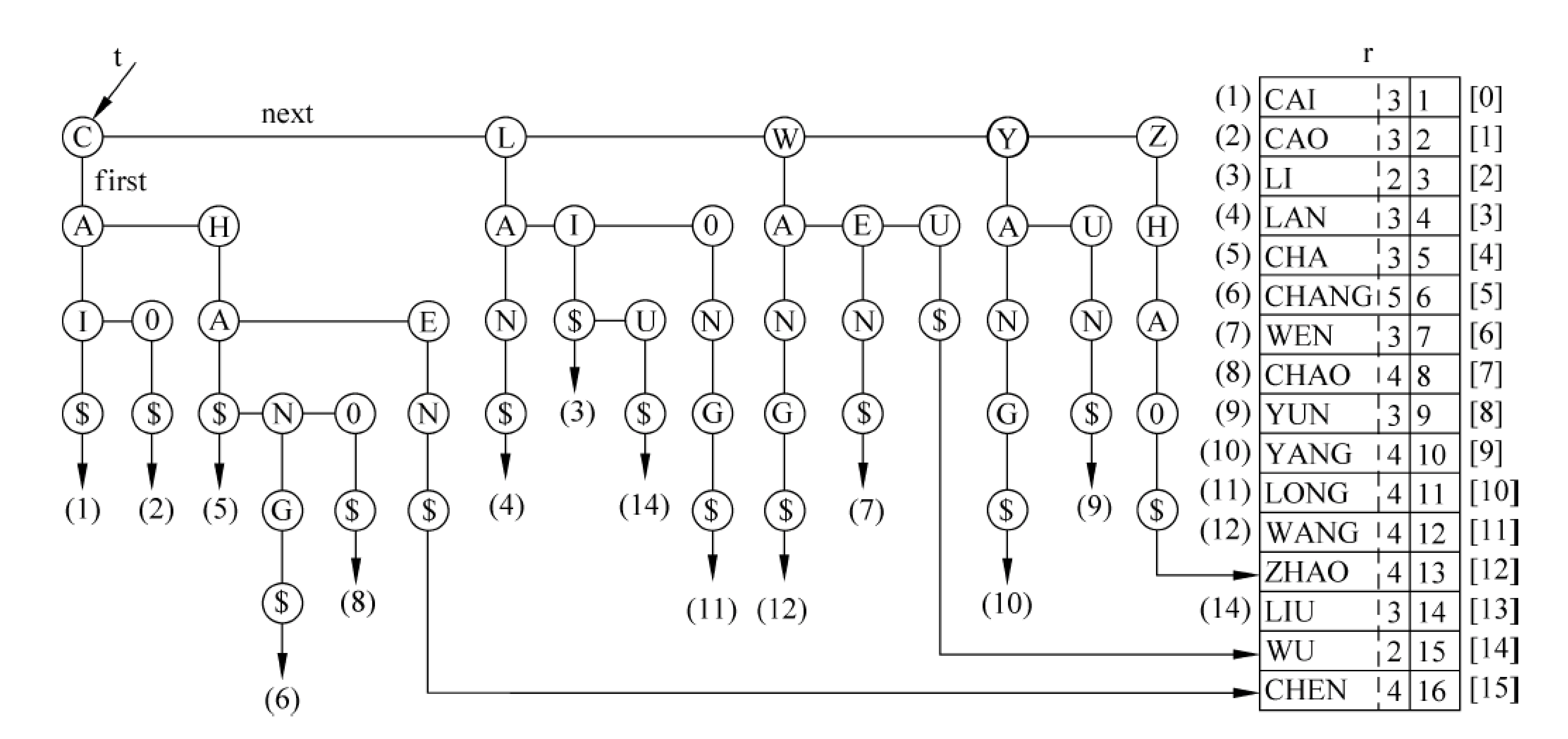



图8-50 运行algo8-7.cpp生成的双链键树t

```
// c8-9.h Trie 键树的存储结构。在教科书第 250 页
typedef struct TrieNode // Trie 树类型(见图 8-51)
{ NodeKind kind;
  union
  { struct
    { KeyType K;
      Record * infoptr;
    }lf; // 叶子结点
    struct
    { TrieNode * ptr[LENGTH]; // LENGTH 为结点的最大度,在主程定义
      int num; // 分支结点的孩子数
    }bh; // 分支结点
  };
}TrieNode, * TrieTree;
```

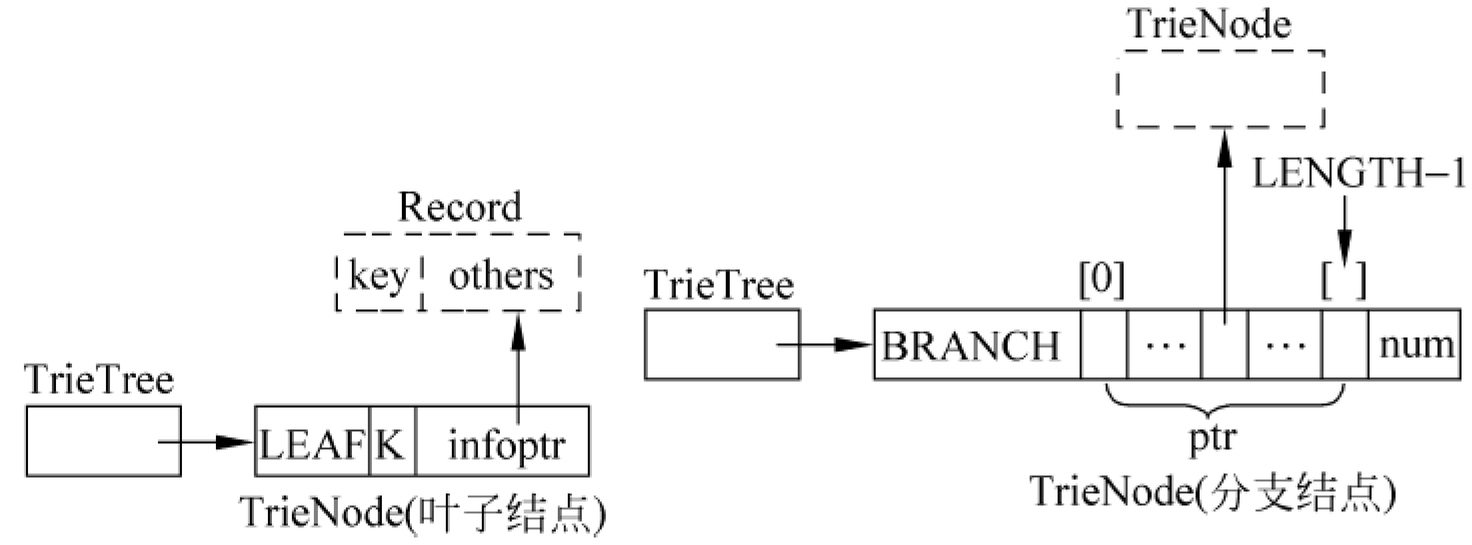



图 8-51 Trie 树存储结构

```
// c8-10.h 对两个字符串型关键字的比较约定为如下的宏定义。在教科书第 215 页
#define EQ(a,b)(!strcmp((a),(b)))
#define LT(a,b)(strcmp((a),(b))<0)
#define LQ(a,b)(strcmp((a),(b))<= 0)
```

```
// bo8-6.cpp Trie 树的基本操作,包括算法 9.16
void InitDSTable(TrieTree &T)
{ // 操作结果:构造一个空的 Trie 树 T(见图 8-52)
  T = NULL;
}
```

T

NULL

图 8-52 空的 Trie 树 T

```
void DestroyDSTable(TrieTree &T)
{ // 初始条件:Trie 树 T 存在。操作结果:销毁 T
  int i,j = 0; // 已处理的结点数
  if(T) // 非空树
  { for(i = 0;j<T->bh.num&&i<LENGTH;i++ )
      if(T->kind == BRANCH&&T->bh.ptr[i]) // T 是分支结点且第 i 个结点不空
      { if(T->bh.ptr[i]->kind == BRANCH) // 第 i 个结点是分支结点
          DestroyDSTable(T->bh.ptr[i]); // 递归销毁第 i 个结点子树
        else // 第 i 个结点是叶子结点
          free(T->bh.ptr[i]); // 释放叶子结点
```

```
        j++; // 已处理的结点数 + 1
      }
    free(T); // 释放根结点
    T = NULL; // 空指针赋 0
  }
}
Record * SearchTrie(TrieTree T,KeyType K)
{ // 在 Trie 树 T 中查找关键字等于 K 的记录。修改算法 9.16(见图 8-53)
  TrieTree p = T;
  int i;
  for(i = 0;p&&p->kind == BRANCH&&i<= K.num;p = p->bh.ptr[ord(K.ch[i++ ])]);
  // 对 K 的每个字符逐个查找,* p 为分支结点,ord()求字符在字母表中序号
  if(p&&p->kind == LEAF&&EQ(p->lf.K.ch,K.ch)) // 查找成功
    return p->lf.infoptr; // 返回关键字等于 K 的记录的地址
  else // 查找不成功
    return NULL; // 返回空
}
void InsertTrie(TrieTree &T,Record* r)
{ // 初始条件:Trie 树 T 存在,r 为待插入的数据元素的指针
  // 操作结果:若 T 中不存在其关键字等于 r->key.ch 的数据元素,则按关键字顺序插 r 到 T 中
  TrieTree p = T,q,ap;
  int i,j,k,n;
  if(!T) // 空树(见图 8-54)
```



图 8-53　调用 SearchTrie()示例



图 8-54　空树调用 InsertTrie()示例

```
  { T = (TrieTree)malloc(sizeof(TrieNode)); // 动态生成 Trie 树结点
    T->kind = BRANCH; // 结点类型为分支
    for(i = 0;i<LENGTH;i++ ) // 指针数组赋初值 NULL
      T->bh.ptr[i] = NULL;
    T->bh.num = 1; // 分支结点的孩子数赋值 1
    p = T->bh.ptr[ord(r->key.ch[0])] = (TrieTree)malloc(sizeof(TrieNode));
    // 在关键字首字符对应的指针处动态生成叶子结点
    p->kind = LEAF; // 结点类型为叶子
    p->lf.K = r->key; // 结点关键字为数据元素的关键字
    p->lf.infoptr = r; // 结点指针指向数据元素的地址
  }
```

```
else // 非空树
{ for(i=0;p&&p->kind==BRANCH&&i<=r->key.num;++i) // 在分支结点中查找插入位置
  { q=p; // q 指向 p 所指结点
    p=p->bh.ptr[ord(r->key.ch[i])]; // p 指向第 i 个关键字所在的结点
  }
  if(!p) // 分支空(见图 8-55)
```

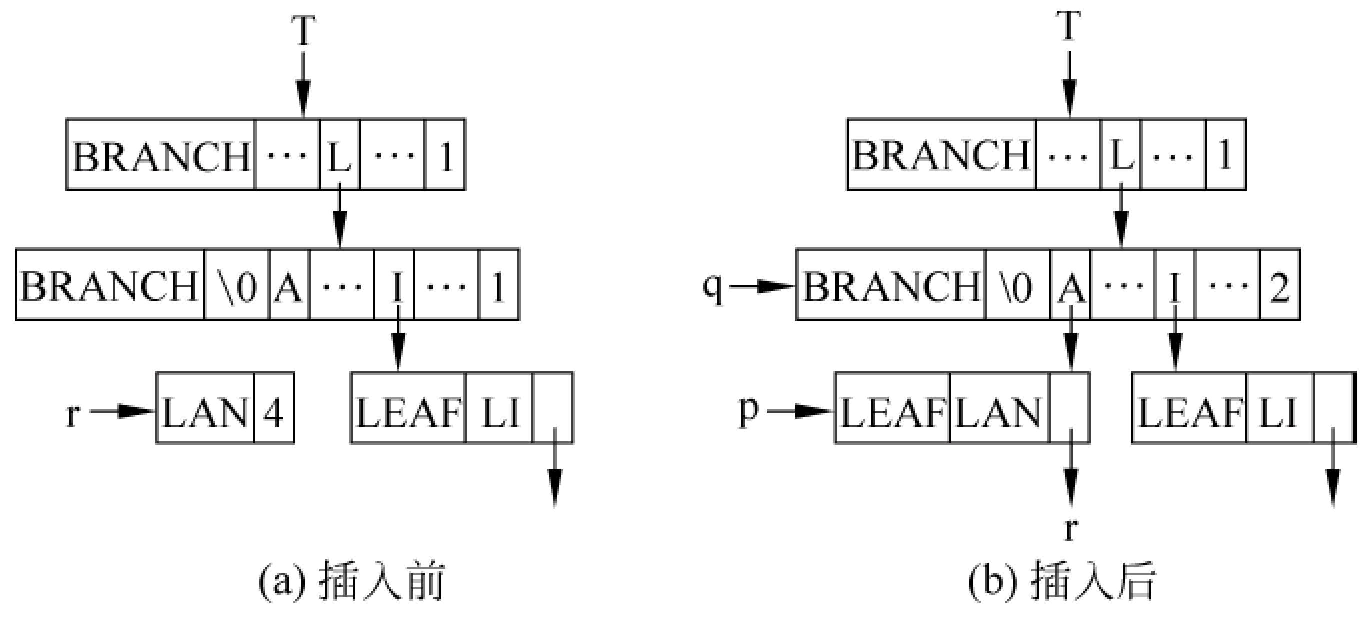



图 8-55　在空分支中插入叶子结点

```
  { p=q->bh.ptr[ord(r->key.ch[i-1])]=(TrieTree)malloc(sizeof(TrieNode));
    // 动态生成 Trie 树结点
    q->bh.num++; // 结点的双亲的孩子数+1
    p->kind=LEAF; // 结点类型为叶子
    p->lf.K=r->key; // 结点关键字为数据元素的关键字
    p->lf.infoptr=r; // 结点指针指向数据元素的地址
  }
  else // p->kind!=BRANCH
  { if(EQ(p->lf.K.ch,r->key.ch)) // T 中存在该关键字
      return; // 不插入返回
    else // 有关键字部分相同的叶子(见图 8-56)
```

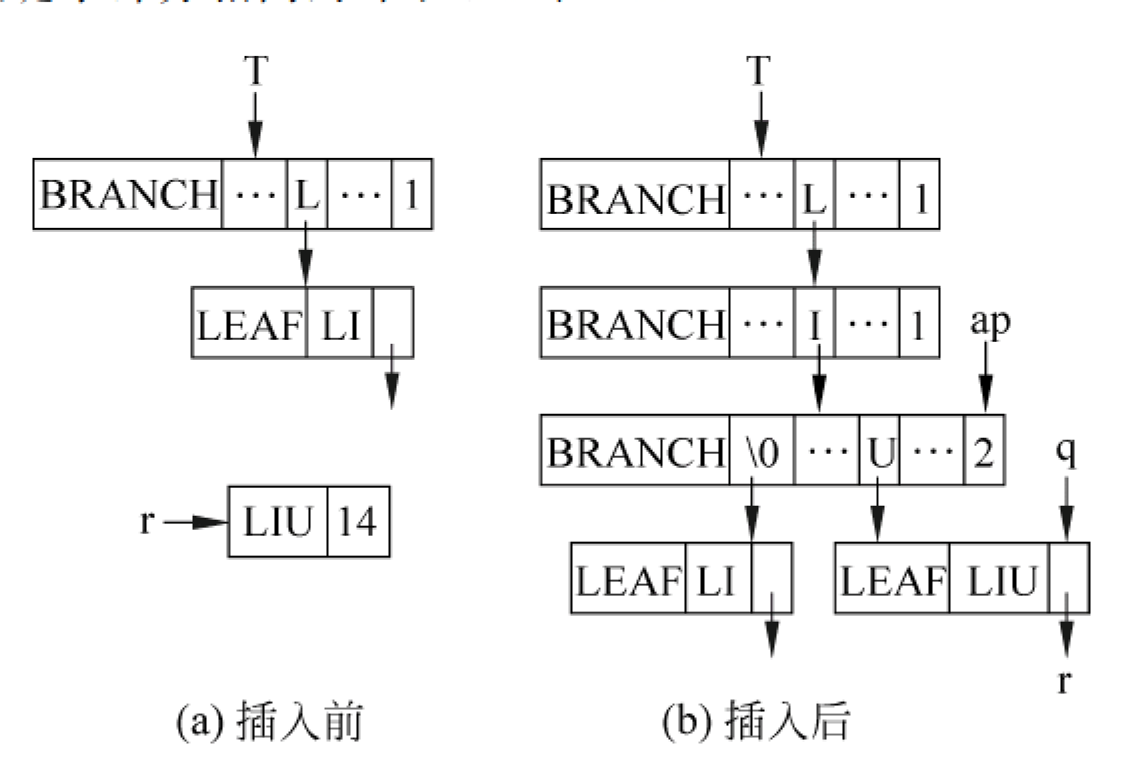



图 8-56　存在关键字部分相同的叶子结点

```
    { for(n=0;r->key.ch[n]==p->lf.K.ch[n];n++); // n 为相同的关键字符数
      n=n-i;
      for(k=0;k<=n;k++)
      { ap=q=q->bh.ptr[ord(r->key.ch[i++ -1])]=(TrieTree)malloc(sizeof
        (TrieNode)); // 动态生成 Trie 树结点
        ap->kind=BRANCH; // 结点类型为分支
```

```
            for(j=0;j<LENGTH;j++)
              ap->bh.ptr[j]=NULL; // 指针数组赋初值 NULL
            ap->bh.num=1; // 分支结点的孩子数赋值 1
          }
          ap->bh.num++; // 分支结点的孩子数加 1
          ap->bh.ptr[ord(p->lf.K.ch[i-1])]=p; // 将 *p 接入 Trie 树中
          q=ap->bh.ptr[ord(r->key.ch[i-1])]=(TrieTree)malloc(sizeof(TrieNode));
          // 动态生成 Trie 树结点
          q->kind=LEAF; // 结点类型为叶子
          q->lf.K=r->key; // 结点关键字为数据元素的关键字
          q->lf.infoptr=r; // 结点指针指向数据元素的地址
        }
      }
    }
}
void TraverseDSTable(TrieTree T,void(*Visit)(Record*))
{ // 初始条件：Trie 树 T 存在，Visit 是对记录指针操作的应用函数
  // 操作结果：按关键字的顺序输出关键字及其对应的记录
  TrieTree p;
  int i,n=9; // 输出 n 个元素后换行
  if(T) // 树 T 不空
    for(i=0;i<LENGTH;i++) // 对 T 的所有的孩子
    { p=T->bh.ptr[i]; // p 指向 T 的第 i 个孩子
      if(p&&p->kind==LEAF) // T 的第 i 个孩子是叶子
      { Visit(p->lf.infoptr); // 访问该结点所指的记录
        count++;
        if(count%n==0)
          printf("\n"); // 输出 n 个元素后换行
      }
      else if(p&&p->kind==BRANCH) // T 的第 i 个孩子是分支
        TraverseDSTable(p,Visit); // 对 T 的第 i 个孩子递归调用 TraverseDSTable()
    }
}
```

```
// algo8-8.cpp 检验 bo8-6.cpp 的程序
#include"c1.h"
#define N 20 // 数组可容纳的数据元素个数
#define LENGTH 27 // 结点的最大度(0+大写英文字母)
struct Others // 记录的其他部分
{ int order; // 整型变量，顺序
};
#define Nil 0 // 定义串结束符为 0
static int count=0; // 调用 TraverseDSTable()的次数，初始为 0
```

```
#include"c8-7.h" // 键树记录的存储结构
#include"c8-4.h" // 记录类型
#include"c8-9.h" // Trie 键树的存储结构
#include"c8-10.h" // 对两个字符串型关键字的比较约定
int ord(char c)
{ if(c == Nil)
    return 0; // 遇字符串结束符返回 0
  else
    return toupper(c) - 'A' + 1; // 英文字母返回其在字母表中的序号(此处不分大小写)
}
#include"bo8-6.cpp" // Trie 树的基本操作,包括算法 9.16
#include"func8-7.cpp" // 包括对键树的输入输出操作
void main()
{
  TrieTree t;
  int i,j = 0; // 数据个数,初始为 0
  KeyType k;
  Record * p,r[N]; // 记录数组
  FILE * f; // 文件指针类型
  InitDSTable(t); // 构造空的 Trie 树 t
  f = fopen("f8-5.txt","r"); // 打开数据文件 f8-5.txt
  do // 将数据文件中的记录依次读入 r 并插入 Trie 树 t 中
  { i = fscanf(f,"%s%d",&r[j].key.ch,&r[j].others.order); // 由文件输入数据给 r[j]
    if(i != - 1) // 输入数据成功
    { r[j].key.num = strlen(r[j].key.ch); // 给 r[j].key.num 域赋值
      p = SearchTrie(t,r[j].key); // 在 Trie 树 t 中查找关键字串等于 r[j].key 的记录
      if(!p) // t 中不存在关键字为 r[j].key 的项
        InsertTrie(t,&r[j++]);
        // 将 r[j]的地址及关键字串插入 Trie 树 t 中,记录数组的数据个数 + 1
      else // 在树 t 中已存在关键字为 r[j].key 的项
        printf("树 t 中已存在关键字为%s 的记录,故(%s,%d)无法插入。\n",
        r[j].key.ch,r[j].key.ch,r[j].others.order);
    }
  }while(!feof(f)&&j<N); // 未到数据文件的结尾且记录数组未满
  fclose(f); // 关闭数据文件
  printf("按关键字符串的顺序遍历树 t:\n");
  TraverseDSTable(t,Visit); // 按关键字符串的顺序遍历 Trie 树 t
  printf("\n 请输入待查找记录的关键字符串:");
  InputKey(k.ch); // 输入待查找记录的关键字符串给 k.ch
  k.num = strlen(k.ch); // 给 k.num 赋值
  p = SearchTrie(t,k); // 在 Trie 树 t 查找关键字等于 k 的记录
  if(p) // 存在该记录
    Visit(p); // 输出该记录
```

```
    else // 不存在该记录
      printf("未找到"); // 输出未找到信息
    printf("\n");
    DestroyDSTable(t); // 销毁 Trie 树 t
}
```

程序运行的结果同 algo8-7.cpp 的运行结果，只是图不同，如图 8-57 所示。

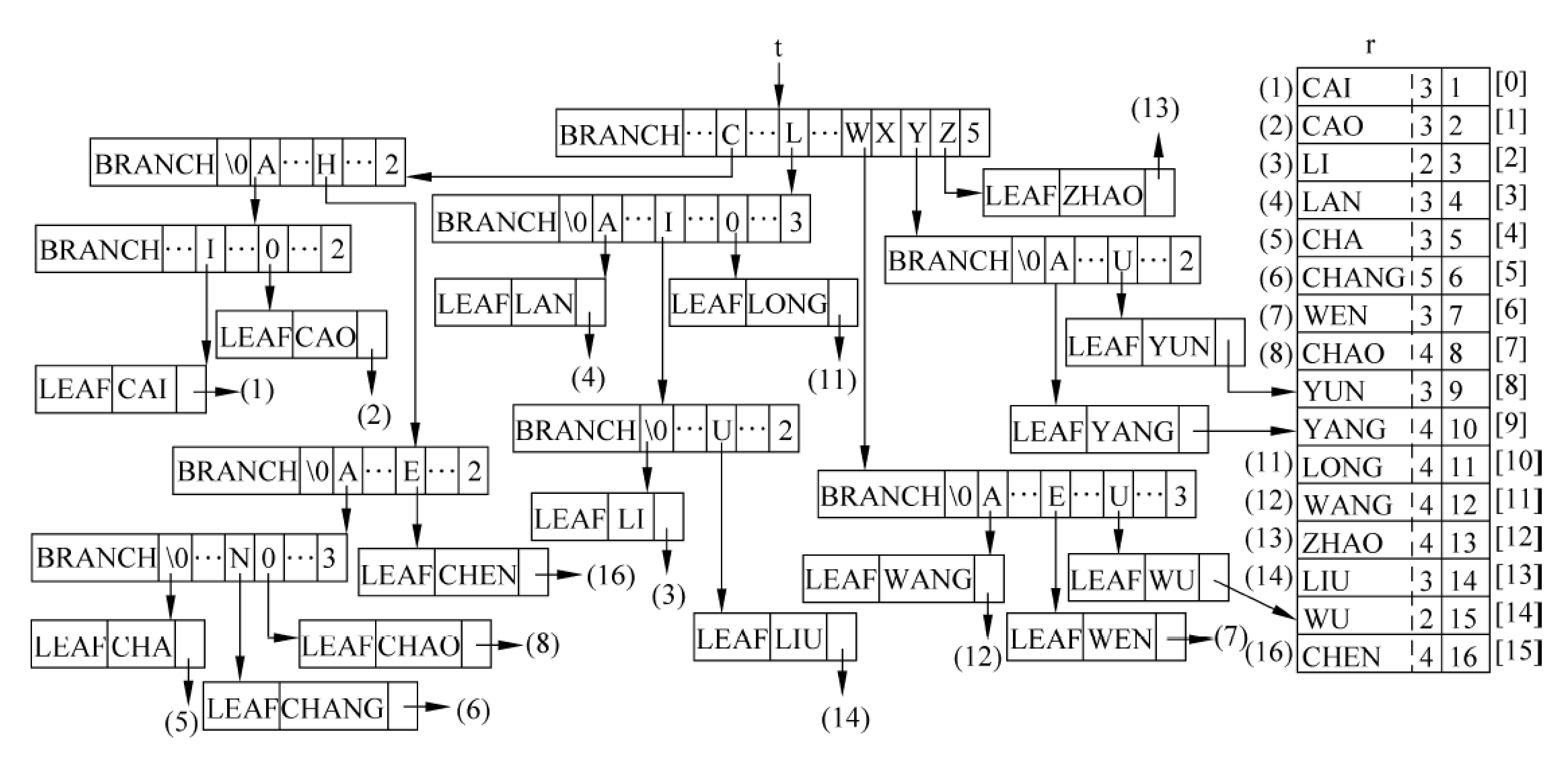



图 8-57　运行 algo8-8.cpp 生成的 Trie 树 t

# 8.3　哈　希　表

## 8.3.1　处理冲突的方法

由于哈希函数是杂凑函数，它有很大的随意性，有可能在一个哈希地址上分配多个数据，这种情况称为冲突。处理冲突，就是把原本分配到一个哈希地址上的多个数据，根据某种原则分配到不同的地址上。

## 8.3.2　哈希表的查找及其分析

```
// c8-11.h 开放定址哈希表的存储结构。在教科书第 259 页
int hashsize[] = {11,19,29,37}; // 哈希表容量递增表，一个合适的素数序列
struct HashTable(见图 8-58)
{ ElemType * elem; // 数据元素存储基址，动态分配数组
  int count; // 当前数据元素个数
  int sizeindex; // hashsize[sizeindex]为当前容量
};
#define SUCCESS 1
```

HashTable　ElemType
elem
count
sizeindex

图 8-58　开放定址哈希表存储结构

```
#define UNSUCCESS 0
#define DUPLICATE -1

// bo8-7.cpp 哈希函数的基本操作,包括算法 9.17 和算法 9.18
void InitHashTable(HashTable &H)
{ // 操作结果:构造一个空的哈希表(见图 8-59)
  int i;
  H.count = 0; // 当前元素个数为 0
  H.sizeindex = 0; // 初始存储容量最小,为 hashsize[0]
  m = hashsize[0]; // 哈希表表长,全局变量
  H.elem = (ElemType *)malloc(m * sizeof(ElemType));
  if(!H.elem)
    exit(OVERFLOW); // 存储分配失败
  for(i = 0;i<m;i++)
    H.elem[i].key = NULL_KEY; // 未填记录的标志
}
```



图 8-59 空的哈希表 H

```
void DestroyHashTable(HashTable &H)
{ // 初始条件:哈希表 H 存在。操作结果:销毁哈希表 H(见图 8-60)
  free(H.elem); // 释放 H.elem 的存储空间
  H.elem = NULL;
  H.count = 0;
  H.sizeindex = 0;
}
```



图 8-60 销毁的哈希表 H

```
unsigned Hash(KeyType K)
{ // 一个简单的哈希函数(m 为表长,全局变量)
  return K % m;
}
int d(int i) // 增量序列函数,在以下 3 行中根据需要选取 1 行,其余 2 行作为注释
{ return i; // 线性探测再散列
//return((i + 1)/2) * ((i + 1)/2) * (int)pow( - 1,i - 1); // 二次探测再散列
//return rand(); // 伪随机探测再散列
}
void collision(KeyType K,int &p,int i)
{ p = (Hash(K) + d(i)) % m; // 开放定址法处理冲突
}
Status SearchHash(HashTable H,KeyType K,int &p,int &c)
{ // 在开放定址哈希表 H 中查找关键字为 K 的元素,若查找成功,以 p 指示待查数据
  // 元素在表中位置,并返回 SUCCESS;否则,以 p 指示插入位置,并返回 UNSUCCESS
  // c 用以计冲突次数,其初值置零,供建表插入时参考。修改算法 9.17
  p = Hash(K); // 求得哈希地址
  while(H.elem[p].key != NULL_KEY && !EQ(K,H.elem[p].key))
  { // 该位置中填有记录.并且与待查找的关键字不相等
    c++; // 计冲突次数 + 1
    if(c<m) // 在 H 中有可能找到插入地址,修改
      collision(K,p,c); // 求得下一探查地址 p
```

```
    else // 在 H 中不可能找到插入地址
      break; // 退出 while 循环
  }
  if EQ(K,H.elem[p].key) // 查找成功
    return SUCCESS; // p 返回待查数据元素位置
  else // 查找不成功
    return UNSUCCESS; // H.elem[p].key == NULL_KEY,p 返回的是插入位置
}
void RecreateHashTable(HashTable&); // 对函数 RecreateHashTable()的声明
Status InsertHash(HashTable &H,ElemType e)
{ // 查找不成功时插入数据元素 e 到开放定址哈希表 H 中,并返回 OK;
  // 若冲突次数过大,则重建哈希表,算法 9.18
  int p,c = 0; // 冲突次数,初始为 0
  if(SearchHash(H,e.key,p,c)) // 查找成功
    return DUPLICATE; // H 中已有与 e 有相同关键字的元素,不再插入
  else if(c<hashsize[H.sizeindex]/2) // 未找到,冲突次数也 c 未达到上限,(c 的阀值可调)
  { H.elem[p] = e; // 在 H 中插入数据元素 e
    ++ H.count; // H 的数据元素个数 + 1
    return OK; // 插入成功
  }
  else // 未找到,但冲突次数 c 已达到上限
  { RecreateHashTable(H); // 重建哈希表
    return UNSUCCESS; // 插入不成功,需重新插入
  }
}
void RecreateHashTable(HashTable &H)
{ // 重建哈希表 H
  int i,count = H.count; // H 中原有数据个数
  ElemType * p, * elem = (ElemType * )malloc(count * sizeof(ElemType));
  // 动态生成存放哈希表 H 原有数据的空间
  p = elem; // p 指向 elem
  for(i = 0;i<m;i++ ) // 将 H 原有的第 1 个数据到最后 1 个数据,保存到 elem 中
    if((H.elem + i) ->key!= NULL_KEY) // H 在该单元有数据
      * p++ = * (H.elem + i); // 将数据依次存入 elem
  H.count = 0; // H 的当前数据元素个数为 0
  H.sizeindex++; // 增大存储容量为下一个序列数
  m = hashsize[H.sizeindex]; // 新的存储容量
  H.elem = (ElemType * )realloc(H.elem,m * sizeof(ElemType));
  // 以新的存储容量重新生成空哈希表 H
  for(i = 0;i<m;i++ )
    H.elem[i].key = NULL_KEY; // 未填记录的标志(初始化)
  for(p = elem;p<elem + count;p++ ) // 将原有的数据按照新的表长插入到重建的哈希表中
    InsertHash(H, * p);
  free(elem); // 释放 elem 的存储空间
}
```

```
void TraverseHash(HashTable H,void(* Visit)(int,ElemType))
{ // 按哈希地址的顺序遍历哈希表 H
  int i;
  printf("哈希地址 0～%d\n",m-1);
  for(i=0;i<m;i++) // 对于整个哈希表 H
    if(H.elem[i].key!=NULL_KEY) // H 在第 i 个单元有数据
      Visit(i,H.elem[i]); // 访问第 i 个数据
}
```

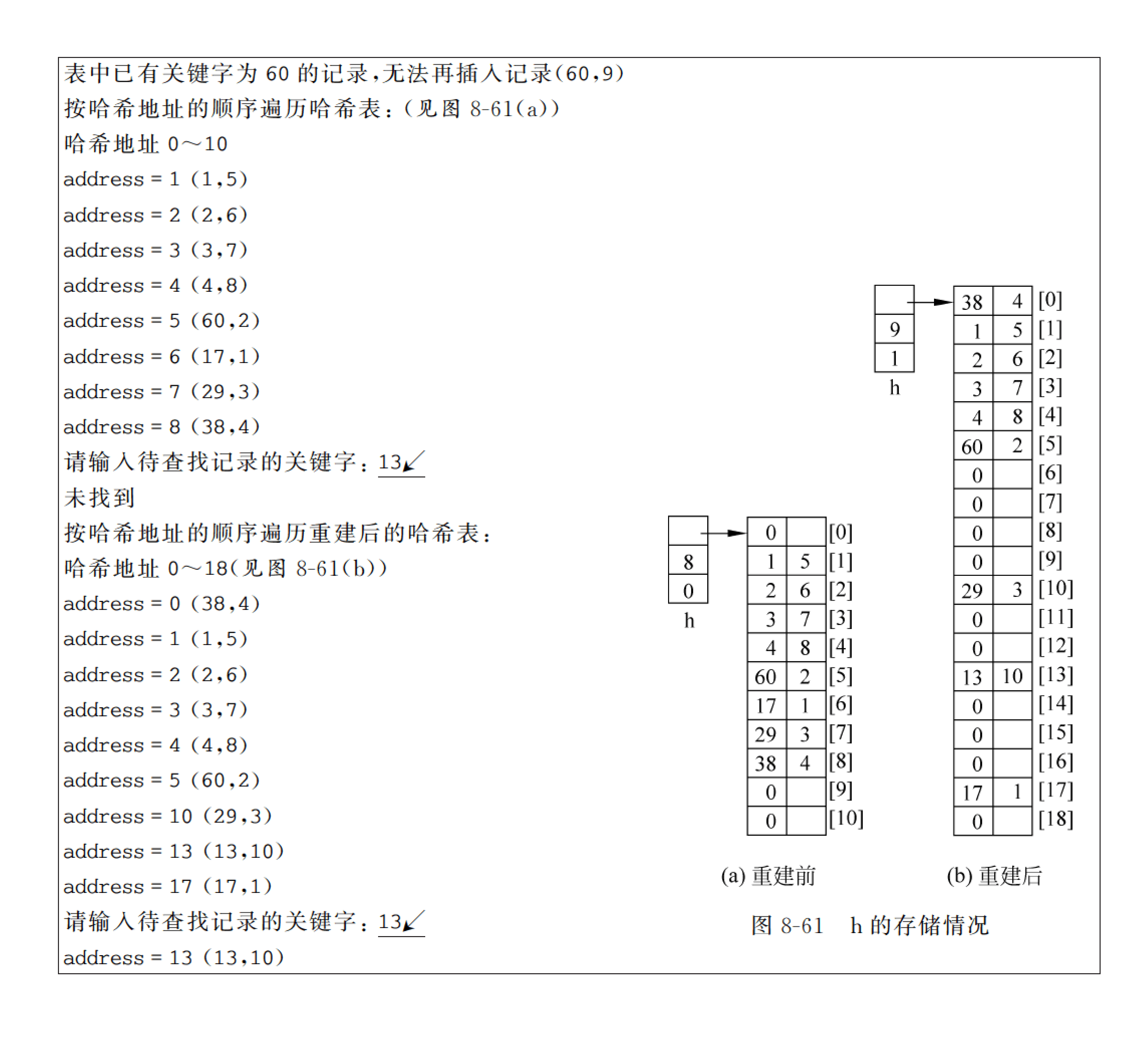
f8-6.txt 内容如下：

```
17 1
60 2
29 3
38 4
1 5
2 6
3 7
4 8
60 9
13 10
```

```
// algo8-9.cpp 检验 bo8-7.cpp 的程序
#include"c1.h"
#define NULL_KEY 0 // 0 为无记录标志
#define N 15 // 数组可容纳的数据元素个数
int m; // 哈希表表长,全局变量
typedef int KeyType; // 定义关键字域为整型
struct ElemType // 数据元素类型
{ KeyType key;
  int order;
};
#include"c8-2.h"
#include"c8-11.h"
#include"bo8-7.cpp"
void Visit(int p,ElemType r)
{ printf("address=%d(%d,%d)\n",p,r.key,r.order);
}
void main()
{
  ElemType r[N]; // 记录数组
  HashTable h;
  int i,n,p=0;
  Status j;
  KeyType k;
```

```
    FILE *f; // 文件指针类型
    f=fopen("f8-6.txt","r"); // 打开数据文件 f8-6.txt
    do // 将数据文件中的记录依次读入记录数组 r
    { i=fscanf(f,"%d%d",&r[p].key,&r[p].order); // 由文件输入数据给 r[p]
      if(i!=-1) // 输入数据成功
        p++;
    }while(!feof(f)&&p<N); // 未到数据文件的结尾且记录数组未满
    fclose(f); // 关闭数据文件
    InitHashTable(h); // 构造一个空的哈希表 h
    for(i=0;i<p-1;i++)
    { j=InsertHash(h,r[i]); // 在 h 中插入前 p-1 个记录(最后 1 个记录不插入)
      if(j==DUPLICATE)
        printf("表中已有关键字为%d的记录,无法再插入记录(%d,%d)\n",
        r[i].key,r[i].key,r[i].order);
    }
    printf("按哈希地址的顺序遍历哈希表:\n");
    TraverseHash(h,Visit); // 按哈希地址的顺序遍历哈希表 h
    printf("请输入待查找记录的关键字:");
    scanf("%d",&k); // 输入待查找记录的关键字
    j=SearchHash(h,k,p,n); // 查找时 n 值无用
    if(j==SUCCESS) // 查找成功
      Visit(p,h.elem[p]); // 输出该纪录
    else // 查找失败
      printf("未找到\n"); // 输出未找到信息
    j=InsertHash(h,r[i]); // 插入最后 1 个记录(需重建哈希表)
    if(j==ERROR) // 重建哈希表
      j=InsertHash(h,r[i]); // 重建哈希表后重新插入
    printf("按哈希地址的顺序遍历重建后的哈希表:\n");
    TraverseHash(h,Visit); // 按哈希地址的顺序遍历哈希表 h
    printf("请输入待查找记录的关键字:");
    scanf("%d",&k); // 输入待查找记录的关键字
    j=SearchHash(h,k,p,n); // 查找时 n 值无用
    if(j==SUCCESS) // 查找成功
      Visit(p,h.elem[p]); // 输出该纪录
    else // 查找失败
      printf("未找到\n"); // 输出未找到信息
    DestroyHashTable(h); // 销毁哈希表 h
}
```

**程序运行结果**(以教科书中图 9.25 为例):



```
表中已有关键字为 60 的记录，无法再插入记录(60,9)
按哈希地址的顺序遍历哈希表：(见图 8-61(a))
哈希地址 0～10
address = 1 (1,5)
address = 2 (2,6)
address = 3 (3,7)
address = 4 (4,8)
address = 5 (60,2)
address = 6 (17,1)
address = 7 (29,3)
address = 8 (38,4)
请输入待查找记录的关键字：13↙
未找到
按哈希地址的顺序遍历重建后的哈希表：
哈希地址 0～18(见图 8-61(b))
address = 0 (38,4)
address = 1 (1,5)
address = 2 (2,6)
address = 3 (3,7)
address = 4 (4,8)
address = 5 (60,2)
address = 10 (29,3)
address = 13 (13,10)
address = 17 (17,1)
请输入待查找记录的关键字：13↙
address = 13 (13,10)
```



图 8-61 h 的存储情况

# 第 9 章

# 内 部 排 序

## 9.1 概　　述

排序就是把一组数据按关键字的大小有规律地排列。经过排序的数据更易于查找。所谓内部排序，就是先把待排序数据都放到内存中，再进行排序。

```
// c9-1.h 待排记录的数据类型。在教科书第 264 页
struct RedType // 记录类型(见图 9-1)
{ KeyType key; // 关键字项，具体类型在主程中定义
  InfoType otherinfo; // 其他数据项，具体类型在主程中定义
};
```

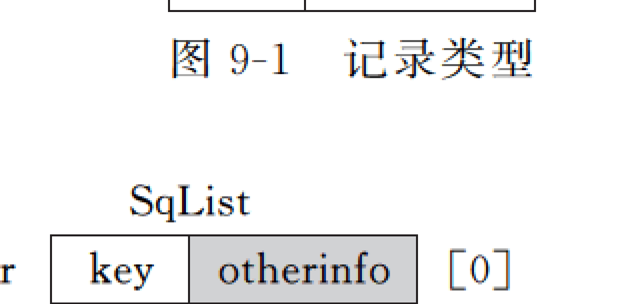



图 9-1　记录类型

```
// c9-2.h 顺序表类型的存储结构。在教科书第 264 页
#define MAX_SIZE 20 // 一个用作示例的小顺序表的最大长度
struct SqList // 顺序表类型(见图 9-2)
{ RedType r[MAX_SIZE + 1]; // r[0]闲置或用作哨兵单元
  int length; // 顺序表长度
};
```

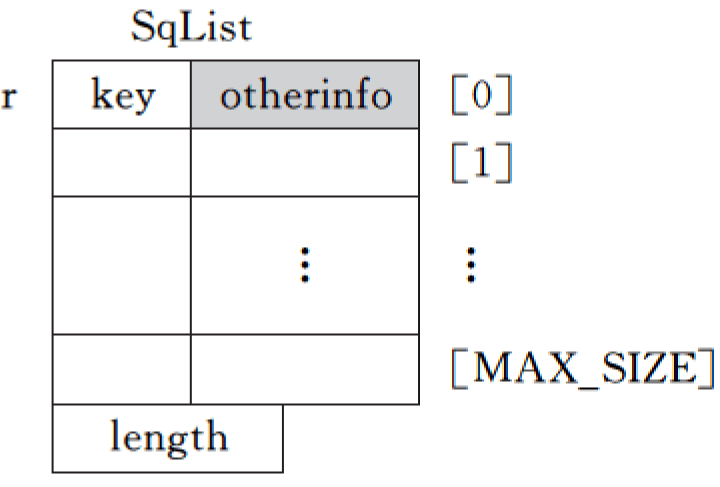



图 9-2　顺序表存储结构

## 9.2 插 入 排 序

### 9.2.1 直接插入排序

```
// bo9-1.cpp 顺序表插入排序的函数(3 个)，包括算法 10.1 和算法 10.2
void InsertSort(SqList &L)
{ // 对顺序表 L 作直接插入排序。算法 10.1
  int i,j;
  for(i = 2;i<= L.length;++i) // 从第 2 个记录到最后一个记录
    if LT(L.r[i].key,L.r[i - 1].key) // 当前记录<前一个记录
    { // 将 L.r[i]插入[1..i - 1]的有序子表中
      L.r[0] = L.r[i]; // 将当前记录复制为哨兵(存入[0]中)
      for(j = i - 1;LT(L.r[0].key,L.r[j].key);--j) // 有序子表从后到前，若哨兵小于记录
```

```
        L.r[j+1]=L.r[j]; // 记录后移 1 个单元
      L.r[j+1]=L.r[0]; // 将哨兵(当前记录)插入到正确位置
    }
}
void BInsertSort(SqList &L)
{ // 对顺序表 L 作折半插入排序。修改算法 10.2
  int i,j,m,low,high;
  for(i=2;i<=L.length;++i) // 从第 2 个记录到最后一个记录
    if LT(L.r[i].key,L.r[i-1].key) // 当前记录<前一个记录,加此句
    { // 将 L.r[i]插入[1..i-1]的有序子表中
      L.r[0]=L.r[i]; // 将 L.r[i]暂存到 L.r[0]
      low=1; // 插入区间的低端
      high=i-1; // 插入区间的高端
      while(low<=high) // 在 r[low..high]中折半查找有序插入的位置
      { m=(low+high)/2; // 折半(中点位置 m)
        if LT(L.r[0].key,L.r[m].key) // 关键字小于中点关键字
          high=m-1; // 插入点在低半区
        else // 关键字大于等于中点关键字
          low=m+1; // 插入点在高半区
      } // low>high,退出 while 循环,[high+1]是插入位置
      for(j=i-1;j>=high+1;--j) // 有序子表从后到前
        L.r[j+1]=L.r[j]; // 记录后移
      L.r[high+1]=L.r[0]; // 插入到[high+1]
    }
}
void P2_InsertSort(SqList &L,int flag)
{ // 2-路插入排序(flag=0)和改进的 2-路插入排序(flag=1,当 L.r[1]是待排序记录中关键字
  // 最小或最大的记录时,仍有优越性)
  int i,j,first,final,mid=0;
  RedType *d;
  d=(RedType*)malloc(L.length*sizeof(RedType)); // 生成 L.length 个记录的临时空间
  d[0]=L.r[1]; // 设 L 的第 1 个记录为 d 中排好序的记录(在位置[0])
  first=final=0; // first、final 分别指示 d 中排好序的记录的第 1 个和最后 1 个记录的位置
  for(i=2;i<=L.length;++i) // 依次将 L 的第 2 个~最后 1 个记录插入 d 中
  { if(flag) // 改进的 2-路插入排序,每次插入前都求 mid
    { if(first>final)
        j=L.length; // j 是调整系数
      else
        j=0;
      mid=(first+final+j)/2%L.length; // d 的中间记录的位置
    }
    if(L.r[i].key<d[mid].key) // 待插记录将插在 d 的前半部分中(flag=0 时,mid=0)
    { j=first; // j 指向 d 的第 1 个记录
      first=(first-1+L.length)%L.length; // first 前移
```

```
      while(L.r[i].key>d[j].key) // 待插记录大于 j 所指记录
      { d[(j-1+L.length)%L.length]=d[j]; // j 所指记录前移
        j=(j+1)%L.length; // j 指向下 1 个记录
      }
      d[(j-1+L.length)%L.length]=L.r[i]; // 移动结束,待插记录插在[j]前
    }
    else // 待插记录将插在后半部分中
    { j=final++; // j 指向当前的最后 1 个记录,final 指向插入后的最后 1 个记录
      while(L.r[i].key<d[j].key) // 待插记录小于 j 所指记录
      { d[(j+1)%L.length]=d[j]; // j 所指记录后移,flag=0 时不必求余
        j=(j-1+L.length)%L.length; // j 指向上 1 个记录,flag=0 时不必求余
      }
      d[(j+1)%L.length]=L.r[i];
      // 待插记录不小于 j 所指记录,插在 j 后,flag=0 时不必求余
    }
  }
  for(i=1;i<=L.length;i++) // 把在 d 中排好序的记录依次赋给 L.r
    L.r[i]=d[(first+i-1)%L.length]; // 线性关系
  free(d); // 释放 d 所指的存储空间
}
// func9-1.cpp 与 KeyType 和 InfoType 均为整型情况配套的输入输出函数
void Print(SqList L) // 输出顺序表
{ int i;
  for(i=1;i<=L.length;i++)
    printf("(%d,%d)",L.r[i].key,L.r[i].otherinfo);
  printf("\n");
}
void InputFromFile(FILE* f,RedType &c) // 从文件输入记录的函数
{ fscanf(f,"%d%d",&c.key,&c.otherinfo);
}
```

数据文件 f9-1.txt 的内容如下：

```
8
49 1
38 2
65 3
97 4
76 5
13 6
27 7
49 8
```

```
// algo9-1.cpp 检验 bo9-1.cpp 的程序
#include"c1.h"
#include"c8-2.h" // 对两个数值型关键字比较的约定
```

```
typedef int KeyType; // 定义关键字的类型为整型
typedef int InfoType; // 定义其他数据项的类型为整型
#include"c9-1.h" // 记录的数据类型
#include"c9-2.h" // 顺序表类型的存储结构
#include"bo9-1.cpp" // 3个顺序表插入排序的函数,包括算法10.1和算法10.2
#include"func9-1.cpp" // 配套的输入输出函数
void main()
{
  FILE *f; // 文件指针类型
  SqList m1,m2,m3,m4; // 4个顺序表变量
  int i;
  f=fopen("f9-1.txt","r"); // 打开数据文件f9-1.txt
  fscanf(f,"%d",&m1.length); // 由数据文件输入数据元素个数给m1.length
  for(i=1;i<=m1.length;i++) // 给m1.r赋值
    InputFromFile(f,m1.r[i]); // 由数据文件输入数据元素的值并赋给m1.r[i]
  fclose(f); // 关闭数据文件
  m2=m3=m4=m1; // 复制顺序表m1到m2、m3、m4,使之与m1相同
  printf("排序前:\n");
  Print(m1); // 输出排序前的顺序表
  InsertSort(m1); // 对m1调用直接插入排序法
  printf("直接插入排序后:\n");
  Print(m1); // 输出排序后的m1
  BInsertSort(m2); // 对m2调用折半插入排序法
  printf("折半插入排序后:\n");
  Print(m2); // 输出排序后的m2
  P2_InsertSort(m3,0); // 对m3调用2-路插入排序法
  printf("2-路插入排序后:\n");
  Print(m3); // 输出排序后的m3
  P2_InsertSort(m4,1); // 对m4调用改进的2-路插入排序法
  printf("改进的2-路插入排序后:\n");
  Print(m4); // 输出排序后的m4
}
```

**程序运行结果**(以教科书中式10-4的数据为例):

```
排序前:
(49,1)(38,2)(65,3)(97,4)(76,5)(13,6)(27,7)(49,8)
直接插入排序后:
(13,6)(27,7)(38,2)(49,1)(49,8)(65,3)(76,5)(97,4)
折半插入排序后:
(13,6)(27,7)(38,2)(49,1)(49,8)(65,3)(76,5)(97,4)
2-路插入排序后:
(13,6)(27,7)(38,2)(49,1)(49,8)(65,3)(76,5)(97,4)
改进的2-路插入排序后:
(13,6)(27,7)(38,2)(49,1)(49,8)(65,3)(76,5)(97,4)
```

## 9.2.2　其他插入排序

折半插入排序和 2-路插入排序在 bo9-1.cpp 中。

```
// c9-3.h 静态链表类型。在教科书第 268 页
#define SIZE 100 // 静态链表容量
struct SLNode // 表结点类型(见图 9-3)
{ RedType rc; // 记录项
  int next; // 指针项
};
struct SLinkListType // 静态链表类型(见图 9-4)
{ SLNode r[SIZE]; // 0 号单元为表头结点
  int length; // 链表当前长度
};
```

SLNode

| rc.key | rc.otherinfo | next |
|---|---|---|

图 9-3　表结点类型

SLinkListType

| | rc.key | rc.otherinfo | next | |
|---|---|---|---|---|
| r | | | | [0] |
| | | | | [1] |
| | | ⋮ | | ⋮ |
| | | | | [SIZE−1] |
| | length | | | |

图 9-4　静态链表存储结构

```
// func9-2.cpp 包括算法 10.18
void Sort(SqList L,int adr[])
{ // 求得 adr[1..L.length],adr[i]为静态链表 L 的第 i 个最小记录的序号
  int i=1,p=L.r[0].next; // p 指向第 1 个记录
  while(p) // 未到链表尾
  { adr[i++]=p; // 将 p 赋给 adr[i],i+1
    p=L.r[p].next; // p 指向下一个记录
  }
}
void Rearrange(SqList &L,int adr[])
{ // adr 给出静态链表 L 的有序次序,即 L.r[adr[i]]是第 i 小的记录。
  // 本算法按 adr 重排 L.r,使其有序。算法 10.18
  int i,j,k;
  for(i=1;i<L.length;++i) // 从顺序表的第 1 个记录到倒数第 2 个记录
    if(adr[i]!=i) // 记录不在正确位置
    { j=i;
      L.r[0]=L.r[i]; // 暂存记录[i]到[0](空出[i]或[j]来)
      while(adr[j]!=i) // 记录不在正确位置
      { // 调整 L.r[adr[j]]的记录到位,直到 adr[j]=i 为止
        k=adr[j];
        L.r[j]=L.r[k]; // 将[j]中应放的记录移来(因[j]空)
        adr[j]=j; // 记录处于正确位置的标志
        j=k; // 新空出的位置赋给 j,以便继续循环调整
      }
      L.r[j]=L.r[0]; // 循环调整完毕,将暂存在[0]的记录赋给 L.r[j]
      adr[j]=j;
```

```
    }
}

// algo9-2.cpp 表插入排序,包括算法 10.3
#include"c1.h"
typedef int KeyType; // 定义关键字的类型为整型
typedef int InfoType; // 定义其他数据项的类型为整型
#include"c9-1.h" // 记录的数据类型
#include"c9-3.h" // 静态链表类型
typedef SLinkListType SqList; // 定义算法 10.18 中的 SqList 类型为 SLinkListType 类型
#include"func9-2.cpp" // 算法 10.18
void PrintL(SLinkListType L) // 按顺序结构输出静态链表的函数
{ int i;
  for(i = 0;i<= L.length;i++)
    printf("i = %d key = %d ord = %d next = %d\n",i,L.r[i].rc.key,
    L.r[i].rc.otherinfo,L.r[i].next);
}
void InputFromFile(FILE* f,RedType &c) // 从文件输入记录的函数
{ fscanf(f,"%d%d",&c.key,&c.otherinfo);
}
void CreatTableFromFile(SLinkListType &SL,FILE* f)
{ // 由数据文件 f 建立未排序的顺序表 SL(next 域不起作用)
  int i;
  fscanf(f,"%d",&SL.length); // 由文件输入表长
  for(i = 1;i<= SL.length;i++)
    InputFromFile(f,SL.r[i].rc); // 依次由文件输入记录到 SL.r[i].rc
}
void MakeTableSorted(SLinkListType &SL)
{ // 使无序的顺序表 SL 成为有序的静态循环链表
  int i,p,q;
  SL.r[0].rc.key = INT_MAX; // 表头结点记录的关键字取最大整数(非降序循环链表的表尾)
  SL.r[0].next = 0; // next 域为 0 形成空循环链表,初始化
  for(i = 1;i<= SL.length;i++) // 依次将 SL 中的数据插入到静态循环链表中
  { q = 0; // q 指向静态链表的表头元素
    p = SL.r[0].next; // p 指向静态链表的第 1 个元素
    while(SL.r[p].rc.key<= SL.r[i].rc.key) // 静态链表向后移
    { // p 所指元素的关键字不大于待插记录的关键字(上行的"="使排序方法是稳定的)
      q = p; // q 指向 p 所指元素
      p = SL.r[p].next; // p 指向下 1 个元素
    } // p 所指元素的关键字大于待插记录的关键字,q 所指元素的关键字不大于待插记录的关键字
    SL.r[q].next = i; // 将当前记录插入静态链表(q 后 p 前)
    SL.r[i].next = p;
```

```
    }
}
void Arrange(SLinkListType &SL)
{ // 根据静态链表 SL 中各结点的指针值调整记录位置,使得 SL 成为非递减有序的顺序表。算法 10.3
  int i,p,q;
  SLNode t;
  p = SL.r[0].next; // p 指示第 1 个记录的当前位置
  for(i = 1;i<SL.length;++i)
  { // SL.r[1..i - 1]中记录已按关键字有序排列,第 i 个记录在 SL 中的当前位置应不小于 i
    while(p<i) // p 所指的记录已排好序
      p = SL.r[p].next; // 继续向后找,跳出已排好序的部分
    q = SL.r[p].next; // q 指示尚未调整的表尾部分
    if(p!= i) // 第 i 个记录不恰好在 p 所指的位置,需移动
    { t = SL.r[p]; // p 和 i 交换记录,使第 i 个记录到位
      SL.r[p] = SL.r[i];
      SL.r[i] = t;
      SL.r[i].next = p; // 指向被移走的记录,[i]已排好序,使得以后可由 while 循环找回 p 所指记录
    }
    p = q; // p 指示尚未调整的表尾部分,为找第 i + 1 个记录作准备
  }
}
void main()
{
  FILE * f; // 文件指针类型
  int * adr,i;
  SLinkListType m1,m2; // 2 个静态链表变量
  f = fopen("f9-1.txt","r"); // 打开数据文件 f9-1.txt
  CreatTableFromFile(m1,f); // 由数据文件 f 建立未排序的顺序表
  fclose(f); // 关闭数据文件
  printf("m1 排序前:\n");
  PrintL(m1); // 输出排序前的顺序表 m1
  MakeTableSorted(m1); // 使无序的顺序表 m1 成为有序的静态链表
  m2 = m1; // 复制静态链表 m1,使 m2 与 m1 相同
  printf("m1、m2 链式有序后:\n");
  PrintL(m1); // 输出链式有序的顺序表 m1
  Arrange(m1); // 将链式有序的顺序表 m1 重排为有序的顺序表
  printf("m1 排序后:\n");
  PrintL(m1); // 输出排序后的顺序表 m1
  adr = (int * )malloc((m2.length + 1) * sizeof(int)); // 动态生成 adr 数组
  Sort(m2,adr); // 求得 adr[1..m2.length],adr[i]为静态链表 m2 的第 i 个最小记录的序号
  for(i = 1;i<= m2.length;i++ ) // 依次输出 adr[i]
  { printf("adr[%d] = %d",i,adr[i]);
    if(i % 4 == 0)
```

```
        printf("\n");
    }
    Rearrange(m2,adr); // 按 adr[]重排 m2.r,使其成为有序的顺序表
    printf("m2 排序后: \n");
    PrintL(m2); // 输出排序后的顺序表 m2
    free(adr); // 释放 adr 所指的存储空间
}
```

**程序运行结果**(以教科书中图 10.3 和图 10.4 的数据为例):

```
m1 排序前:(见图 9-5)
i = 0 key = 29793 ord = 8293 next = 26980
i = 1 key = 49 ord = 1 next = 14368
i = 2 key = 38 ord = 2 next = 28265
i = 3 key = 65 ord = 3 next = 25376
i = 4 key = 97 ord = 4 next = 25927
i = 5 key = 76 ord = 5 next = 25972
i = 6 key = 13 ord = 6 next = 25445
i = 7 key = 27 ord = 7 next = 14386
i = 8 key = 49 ord = 8 next = 26988
m1、m2 链式有序后:(见图 9-6)
i = 0 key = 32767 ord = 8293 next = 6
i = 1 key = 49 ord = 1 next = 8
i = 2 key = 38 ord = 2 next = 1
i = 3 key = 65 ord = 3 next = 5
i = 4 key = 97 ord = 4 next = 0
i = 5 key = 76 ord = 5 next = 4
i = 6 key = 13 ord = 6 next = 7
i = 7 key = 27 ord = 7 next = 2
i = 8 key = 49 ord = 8 next = 3
m1 排序后:(见图 9-7)
i = 0 key = 32767 ord = 8293 next = 6
i = 1 key = 13 ord = 6 next = 6
i = 2 key = 27 ord = 7 next = 7
i = 3 key = 38 ord = 2 next = 7
i = 4 key = 49 ord = 1 next = 6
i = 5 key = 49 ord = 8 next = 8
i = 6 key = 65 ord = 3 next = 7
i = 7 key = 76 ord = 5 next = 8
i = 8 key = 97 ord = 4 next = 0
adr[1] = 6 adr[2] = 7 adr[3] = 2 adr[4] = 1
adr[5] = 8 adr[6] = 3 adr[7] = 5 adr[8] = 4
```

m1

| | | | |
|---|---|---|---|
| | | | [0] |
| 49 | 1 | | [1] |
| 38 | 2 | | [2] |
| 65 | 3 | | [3] |
| 97 | 4 | | [4] |
| 76 | 5 | | [5] |
| 13 | 6 | | [6] |
| 27 | 7 | | [7] |
| 49 | 8 | | [8] |
| | ⋮ | | ⋮ |
| | | | [99] |
| 8 | | | |

图 9-5　m1 排序前

m1、m2

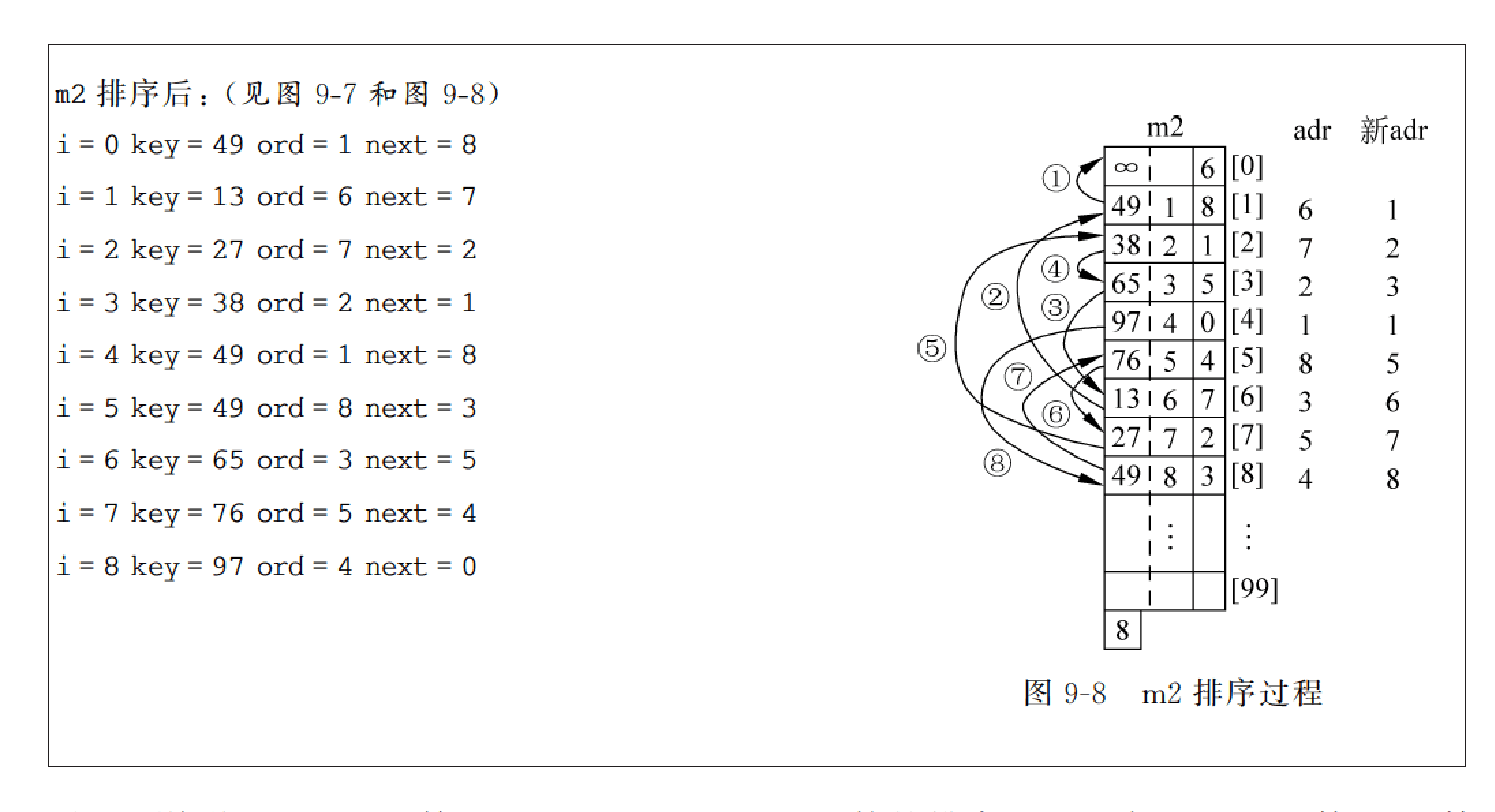
| | | | |
|---|---|---|---|
| ∞ | | 6 | [0] |
| 49 | 1 | 8 | [1] |
| 38 | 2 | 1 | [2] |
| 65 | 3 | 5 | [3] |
| 97 | 4 | 0 | [4] |
| 76 | 5 | 4 | [5] |
| 13 | 6 | 7 | [6] |
| 27 | 7 | 2 | [7] |
| 49 | 8 | 3 | [8] |
| | ⋮ | | ⋮ |
| | | | [99] |
| 8 | | | |

图 9-6　m1、m2 链式有序后

m1、m2

| | | | |
|---|---|---|---|
| | | | [0] |
| 13 | 6 | | [1] |
| 27 | 7 | | [2] |
| 38 | 2 | | [3] |
| 49 | 1 | | [4] |
| 49 | 8 | | [5] |
| 65 | 3 | | [6] |
| 76 | 5 | | [7] |
| 97 | 4 | | [8] |
| | ⋮ | | ⋮ |
| | | | [99] |
| 8 | | | |

图 9-7　m1、m2 排序后



```
m2 排序后：(见图 9-7 和图 9-8)
i = 0 key = 49 ord = 1 next = 8
i = 1 key = 13 ord = 6 next = 7
i = 2 key = 27 ord = 7 next = 2
i = 3 key = 38 ord = 2 next = 1
i = 4 key = 49 ord = 1 next = 8
i = 5 key = 49 ord = 8 next = 3
i = 6 key = 65 ord = 3 next = 5
i = 7 key = 76 ord = 5 next = 4
i = 8 key = 97 ord = 4 next = 0
```



图 9-8 m2 排序过程

图 9-8 说明了 m2 调用算法 10.8，Rearrange()函数的排序过程：当 adr[1]不等于 1(第 1 个记录没有到位)时，先将占据[1]的记录移到空位[0](步骤①)，再将[adr[1]]移到[1](步骤②)，修改 adr[1]等于 1，标志第 1 个记录已经到位。这时，第 1 个记录原先所占的位置又空了，继续前面的步骤，在空位中移入合适的记录(步骤③～⑧)。步骤⑧将[4]中的 97 移到[8]后，根据 adr[4]＝1，应将[1]中的记录移到[4]。但此时 adr[1]＝1，说明[1]中的记录已经移入合适的记录了，原[1] 中的记录必定在[0]。将[0]中的记录移入[4]，完成了一个调整记录位置的小循环。函数 Rearrange()经过这样一个 while 循环，并不能保证所有记录都移入正确的位置。再到 while 循环外面的 for 循环中找到下一个未到正确位置的记录，继续这样的调整，直至所有记录都排到正确位置。

### 9.2.3 希尔排序

```
// algo9-3.cpp 希尔排序，包括算法 10.4 和算法 10.5
#include<stdio.h>
#include"c8-2.h" // 对两个数值型关键字比较的约定
typedef int KeyType; // 定义关键字的类型为整型
typedef int InfoType; // 定义其他数据项的类型为整型
#include"c9-1.h" // 记录的数据类型
#include"c9-2.h" // 顺序表类型的存储结构
#include"func9-1.cpp" // 配套的输入输出函数
void Print1(SqList L) // 输出顺序表 L 的关键字
{ int i;
  for(i = 1;i<= L.length;i++ )
    printf("%d",L.r[i].key);
  printf("\n");
}
void ShellInsert(SqList &L,int dk)
{ // 对顺序表 L 作一趟希尔插入排序。本算法和一趟直接插入排序相比，作了以下修改：
```

```
  // 1.前后记录位置的增量是 dk,而不是 1;
  // 2.r[0]只是暂存单元,不是哨兵。当 j<=0 时,插入位置已找到。算法 10.4
  int i,j;
  for(i=dk+1;i<=L.length;++i) // 从与第 1 个记录相差增量 dk 的记录到表尾
    if LT(L.r[i].key,L.r[i-dk].key) // 关键字小于前面记录的(按增量)
    { // 以下将 L.r[i]插入有序增量子表
      L.r[0]=L.r[i]; // 当前记录暂存在 L.r[0]
      for(j=i-dk;j>0&&LT(L.r[0].key,L.r[j].key);j-=dk)
        L.r[j+dk]=L.r[j]; // 记录后移,查找插入位置
      L.r[j+dk]=L.r[0]; // 插入
    }
}
void ShellSort(SqList &L,int dlta[],int t)
{ // 按增量序列 dlta[0..t-1]对顺序表 L 作希尔排序。算法 10.5
  int k;
  for(k=0;k<t;++k) // 对所有增量序列
  { ShellInsert(L,dlta[k]); // 一趟增量为 dlta[k]的希尔插入排序
    printf("dlta[%d]=%d,第%d 趟排序结果(仅输出关键字):",k,dlta[k],k+1);
    Print1(L); // 输出顺序表 L 的关键字
  }
}
#define N 10 // 记录数组长度
#define T 3 // 增量序列数组长度
void main()
{
  RedType d[N]={{49,1},{38,2},{65,3},{97,4},{76,5},{13,6},{27,7},{49,8},
  {55,9},{4,10}}; // 记录数组
  SqList m; // 顺序表
  int i,dt[T]={5,3,1}; // 增量序列数组(由大到小,最后一项的值必为 1)
  for(i=0;i<N;i++) // 将数组 d 赋给顺序表 m
    m.r[i+1]=d[i];
  m.length=N;
  printf("希尔排序前:");
  Print(m); // 输出排序前的顺序表 m
  ShellSort(m,dt,T); // 按增量序列 dt[0..T-1]对顺序表 m 作希尔排序
  printf("希尔排序后:");
  Print(m); // 输出排序后的顺序表 m
}
```

**程序运行结果**(以教科书中图 10.5 的数据为例):

```
希尔排序前:(49,1)(38,2)(65,3)(97,4)(76,5)(13,6)(27,7)(49,8)(55,9)(4,10)
dlta[0]=5,第 1 趟排序结果(仅输出关键字):13 27 49 55 4 49 38 65 97 76
dlta[1]=3,第 2 趟排序结果(仅输出关键字):13 4 49 38 27 49 55 65 97 76
dlta[2]=1,第 3 趟排序结果(仅输出关键字):4 13 27 38 49 49 55 65 76 97
希尔排序后:(4,10)(13,6)(27,7)(38,2)(49,8)(49,1)(55,9)(65,3)(76,5)(97,4)
```

# 9.3 快速排序

数据文件 f9-2.txt 的内容如下：

```
49 38 65 97 76 13 27 49
```

```
// algo9-4.cpp 调用起泡排序的程序
#include"c1.h"
#define N 20 // 数组长度
void bubble_sort(int a[],int n)
{ // 将 a 中 n 个整数重新排列成自小至大的有序序列(在教科书第 16 页)
  int i,j,t;
  Status change; // 调整的标志
  for(i=n-1,change=TRUE;i>=1&&change;--i) // 由后到前调整,change=FALSE 时终止循环
  { change=FALSE; // 本次循环未调整的标志
    for(j=0;j<i;++j) // 从第 1 个到倒数第 2 个
      if(a[j]>a[j+1]) // 前面的大于后面的
      { t=a[j]; // 前后交换
        a[j]=a[j+1];
        a[j+1]=t;
        change=TRUE; // 设置调整的标志
      }
  }
}
void Print2(int r[],int n)
{ // 输出数组 r 的前 n 个数
  int i;
  for(i=0;i<n;i++)
    printf("%d",r[i]);
  printf("\n");
}
void main()
{
  FILE *f; // 文件指针类型
  int i=0,j,d[N];
  f=fopen("f9-2.txt","r"); // 打开数据文件 f9-2.txt
  do
  { j=fscanf(f,"%d",&d[i++]); // 将文件 f9-2.txt 中的整数依次赋给 d[i]
  }while(j!=EOF); // 未到文件尾
  fclose(f); // 关闭 f 所指的文件,f 不再指向 f9-2.txt 文件
  i--; // i 为读入的数据个数
  printf("排序前:\n");
  Print2(d,i); // 输出数组 d 排序前的前 i 个数
  bubble_sort(d,i); // 对数组 d 的前 i 个数调用起泡排序法
```

```
  printf("排序后：\n");
  Print2(d,i); // 输出数组 d 排序后的前 i 个数
}
```

**程序运行结果**(以教科书中图 10.6 的数据为例)：

```
排序前：
49 38 65 97 76 13 27 49
排序后：
13 27 38 49 49 65 76 97
```

```
// bo9-2.cpp 快速排序的函数,包括算法 10.7 和算法 10.8
void QSort(SqList &L,int low,int high)
{ // 对顺序表 L 中的子序列 L.r[low..high]作快速排序。算法 10.7
  int pivotloc;
  if(low<high) // 子序列长度大于 1
  { pivotloc = Partition(L,low,high);
    // 将 L.r[low..high]按关键字一分为二,pivotloc 是枢轴位置
    QSort(L,low,pivotloc - 1); // 对关键字小于枢轴关键字的低子表递归快速排序
    QSort(L,pivotloc + 1,high); // 对关键字大于枢轴关键字的高子表递归快速排序
  }
}
void QuickSort(SqList &L)
{ // 对顺序表 L 作快速排序。算法 10.8
  QSort(L,1,L.length); // 对整个顺序表 L 作快速排序
}

// func9-3.cpp 算法 10.6(a)
int Partition(SqList &L,int low,int high)
{ // 交换顺序表 L 中子表 L.r[low..high]的记录,使枢轴记录到位,
  // 并返回其所在位置,此时在它之前(后)的记录均不大(小)于它。算法 10.6(a)
  RedType t;
  KeyType pivotkey; // 枢轴关键字
  pivotkey = L.r[low].key; // 用子表的第 1 个记录作初始枢轴记录
  while(low<high) // 未分类的区间大于 0
  { // 从表的两端交替地向中间扫描
    while(low<high&&L.r[high].key>= pivotkey) // 高端记录的关键字大于枢轴关键字
      --high; // 高端向低移,继续比较
    t = L.r[low]; // 将比枢轴关键字小的记录交换到低端,枢轴到高端
    L.r[low] = L.r[high];
    L.r[high] = t;
    while(low<high&&L.r[low].key<= pivotkey) // 低端记录的关键字小于枢轴关键字
      ++low; // 低端向高移,继续比较
    t = L.r[low]; // 将比枢轴关键字大的记录交换到高端,枢轴到低端
    L.r[low] = L.r[high];
    L.r[high] = t;
  }
```

```
  return low; // 返回枢轴所在位置
}

// func9-4.cpp 算法 10.6(b)(算法 10.6(a)的改进)
int Partition(SqList &L,int low,int high)
{ // 交换顺序表 L 中子表 r[low..high]的记录,枢轴记录到位,并返回其
  // 所在位置,此时在它之前(后)的记录均不大(小)于它。算法 10.6(b)
  KeyType pivotkey; // 枢轴关键字
  pivotkey = L.r[low].key; // 用子表的第 1 个记录作初始枢轴记录
  L.r[0] = L.r[low]; // 将枢轴记录保存到[0],改进处
  while(low<high) // 未分类的区间大于 0
  { // 从表的两端交替地向中间扫描
    while(low<high&&L.r[high].key>= pivotkey) // 高端记录的关键字大于枢轴关键字
      --high; // 高端向低移,继续比较
    L.r[low] = L.r[high]; // 将比枢轴关键字小的记录移到低端,枢轴在[0]不动,改进处
    while(low<high&&L.r[low].key<= pivotkey) // 低端记录的关键字小于枢轴关键字
      ++low; // 低端向高移,继续比较
    L.r[high] = L.r[low]; // 将比枢轴关键字大的记录移到高端,枢轴在[0]不动,改进处
  }
  L.r[low] = L.r[0]; // 枢轴记录到位,改进处
  return low; // 返回枢轴位置
}

// algo9-5.cpp 调用算法 10.7、算法 10.8、算法 10.6(a)和算法 10.6(b)的程序
#include<stdio.h>
typedef int KeyType; // 定义关键字的类型为整型
typedef int InfoType; // 定义其他数据项的类型为整型
#include"c9-1.h" // 记录的数据类型
#include"c9-2.h" // 顺序表类型的存储结构
#include"func9-1.cpp" // 配套的输入输出函数
#include"func9-3.cpp" // 算法 10.6(a),函数 Partition(),此 2 行任取 1 行,运行结果相同
//#include"func9-4.cpp" // 算法 10.6(b),函数 Partition()
#include"bo9-2.cpp" // 快速排序的函数,包括算法 10.7 和算法 10.8
void main ()
{
  FILE *f; // 文件指针类型
  SqList m; // 顺序表变量
  int i;
  f = fopen("f9-1.txt","r"); // 打开数据文件 f9-1.txt
  fscanf(f,"%d",&m.length); // 由数据文件输入数据元素个数给 m.length
  for(i = 1;i<= m.length;i++ ) // 给 m.r 赋值
    InputFromFile(f,m.r[i]); // 由数据文件输入数据元素的值并赋给 m.r[i]
  fclose(f); // 关闭数据文件
  printf("排序前:\n");
  Print(m); // 输出排序前的顺序表 m
```

```
  QuickSort(m); // 对 m 调用快速排序法
  printf("排序后:\n");
  Print(m); // 输出排序后的顺序表 m
}
```

**程序运行结果**(以教科书中图 10.7 的数据为例):

```
排序前:
(49,1)(38,2)(65,3)(97,4)(76,5)(13,6)(27,7)(49,8)
排序后:
(13,6)(27,7)(38,2)(49,1)(49,8)(65,3)(76,5)(97,4)
```

# 9.4 选择排序

```
// bo9-3.cpp 选择排序算法,包括算法 10.9、算法 10.10 和算法 10.11
int SelectMinKey(SqList L,int i)
{ // 返回在 L.r[i..L.length]中 key 最小的记录的序号
  int j,k = i; // 初始设[i]的关键字为最小,存 i 于 k
  KeyType min = L.r[i].key; // [i]的关键字存于 min
  for(j = i + 1;j<= L.length;j++ ) // 依次与其他记录比较
    if(L.r[j].key<min) // 找到更小的关键字
    { k = j; // 分别将更小的赋给 k 和 min
      min = L.r[j].key;
    }
  return k; // 返回序号
}
void SelectSort(SqList &L)
{ // 对顺序表 L 作简单选择排序。算法 10.9
  int i,j;
  RedType t;
  for(i = 1;i<L.length;++i) //   选择第 i 小的记录,并交换到位
  { j = SelectMinKey(L,i); // 在 L.r[i..L.length]中选择 key 最小的记录
    if(i!= j)
    { t = L.r[i]; // 与第 i 个记录交换,使得第 i 小的存于[i]
      L.r[i] = L.r[j];
      L.r[j] = t;
    }
  }
}
void TreeSort(SqList &L)
{ // 树形选择排序
  int i,n;
  RedType *t;
  t = (RedType *)malloc((2 * L.length - 1) * sizeof(RedType)); // 二叉树采用顺序存储结构
```

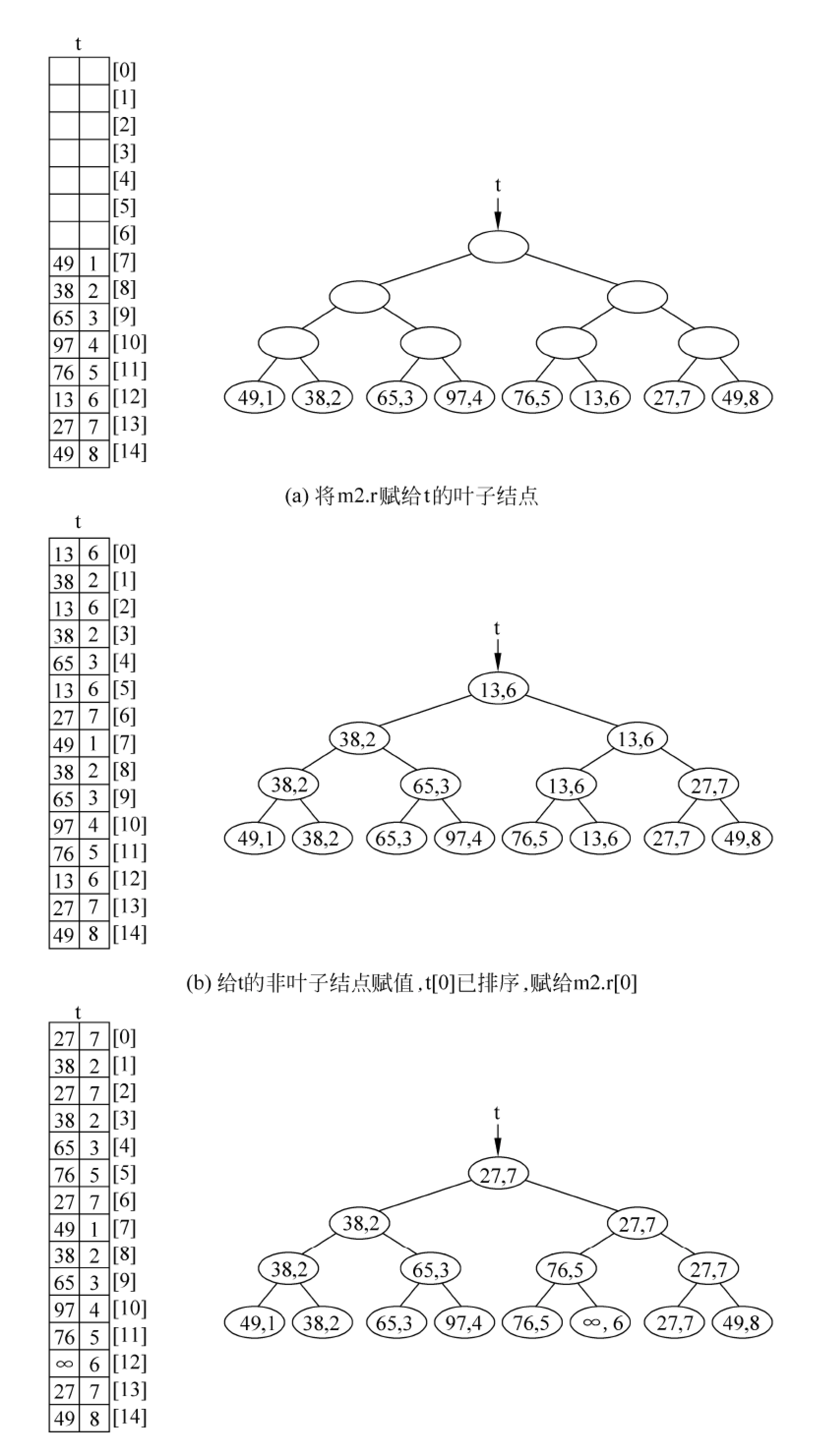
```
  for(i=1;i<=L.length;i++) // 将 L.r 赋给叶子结点(从上到下,从左到右)
    t[L.length-2+i]=L.r[i];
  for(i=L.length-2;i>=0;i--) // 由后向前给非叶子结点赋值
    t[i]=t[2*i+1].key<=t[2*i+2].key?t[2*i+1]:t[2*i+2];
    // 非叶子结点的值为其左右孩子中关键字小的
  for(i=0;i<L.length;i++)
  { L.r[i+1]=t[0]; // 将当前最小值赋给 L.r[i]
    n=0; // 根结点序号
    do // 沿树根按层找结点 t[0]在叶子中的序号 n
    { n=t[2*n+1].key==t[n].key?2*n+1:2*n+2;
    }while(n<L.length-1);
    t[n].key=INT_MAX; // 将此叶子结点的关键字赋无穷大
    while(n) // n 不是根结点
    { n=(n+1)/2-1; // 序号为 n 的结点的双亲结点序号
      t[n]=t[2*n+1].key<=t[2*n+2].key?(t[2*n+1]):(t[2*n+2]);
      // 非叶子结点的值为其左右孩子中关键字小的
    }
  }
  free(t); // 释放 t 所指的存储空间
}
void HeapAdjust(HeapType &H,int s,int m)
{ // 已知 H.r[s..m]中记录的关键字除 H.r[s].key 之外均满足大顶堆的定义,本函数
  // 调整 H.r[s]的关键字,使 H.r[s..m]中记录的关键字均满足大顶堆的定义。算法 10.10
  int j;
  H.r[0]=H.r[s]; // 利用 H 的空闲结点存储待调整记录,修改
  for(j=2*s;j<=m;j*=2) // j 指向待调整记录[s]的左孩子,沿 key 较大的孩子结点向下筛选
  { if(j<m&&LT(H.r[j].key,H.r[j+1].key)) // 左孩子的关键字<右孩子的关键字
      ++j; // j 指向[s]的右孩子
    if(!LT(H.r[0].key,H.r[j].key)) // [s]的关键字不小于[j]的关键字,修改
      break; // 不必再调整,跳出 for 循环
    H.r[s]=H.r[j]; // 否则,[j]为大顶,插入[s]
    s=j; // [s]的位置向下移到[j](原左或右孩子处)
  }
  H.r[s]=H.r[0]; // 将待调整的记录插入[s],修改
}
void HeapSort(HeapType &H)
{ // 对顺序表 H 进行堆排序。算法 10.11
  int i;
  for(i=H.length/2;i>0;--i) // 从最后一个非叶子结点到第 1 个结点
    HeapAdjust(H,i,H.length); // 调整 H.r[i],使 H.r[i..H.length]成为大顶堆
  for(i=H.length;i>1;--i)
  { H.r[0]=H.r[1]; // 将堆顶记录[1]和未完全排序的 H.r[1..i]中的最后一个记录[i]交换
    H.r[1]=H.r[i];
    H.r[i]=H.r[0];
    HeapAdjust(H,1,i-1); // 调整 H.r[1],使 H.r[1..i-1]重新成为大顶堆
```

```
  }
}

// algo9-6.cpp 调用算法 10.9、算法 10.10 和算法 10.11 的程序
#include"c1.h"
#include"c8-2.h" // 对两个数值型关键字比较的约定
typedef int KeyType; // 定义关键字的类型为整型
typedef int InfoType; // 定义其他数据项的类型为整型
#include"c9-1.h" // 记录的数据类型
#include"c9-2.h" // 顺序表类型的存储结构
#include"func9-1.cpp" // 配套的输入输出函数
typedef SqList HeapType; // 定义堆为顺序表存储结构。在教科书第 281 页
#include"bo9-3.cpp" // 选择排序的函数,包括算法 10.9、算法 10.10 和算法 10.11
void main()
{
  FILE *f; // 文件指针类型
  SqList m1,m2,m3; // 3 个顺序表变量
  int i;
  f=fopen("f9-1.txt","r"); // 打开数据文件 f9-1.txt
  fscanf(f,"%d",&m1.length); // 由数据文件输入数据元素个数给 m1.length
  for(i=1;i<=m1.length;i++) // 给 m1.r 赋值
    InputFromFile(f,m1.r[i]); // 由数据文件输入数据元素的值并赋给 m1.r[i]
  fclose(f); // 关闭数据文件
  m2=m3=m1; // 复制顺序表,使 m2、m3 与 m1 相同
  printf("排序前:\n");
  Print(m1); // 输出排序前的顺序表
  SelectSort(m1); // 对 m1 调用简单选择排序法
  printf("简单选择排序后:\n");
  Print(m1); // 输出排序后的 m1
  TreeSort(m2); // 对 m2 调用树形选择排序法
  printf("树形选择排序后:\n");
  Print(m2); // 输出排序后的 m2
  HeapSort(m3); // 对 m3 调用堆排序法
  printf("堆排序后:\n");
  Print(m3); // 输出排序后的 m3
}
```

**程序运行结果:**

```
排序前:
(49,1)(38,2)(65,3)(97,4)(76,5)(13,6)(27,7)(49,8)
简单选择排序后:
(13,6)(27,7)(38,2)(49,1)(49,8)(65,3)(76,5)(97,4)
树形选择排序后:(见图 9-9)
(13,6)(27,7)(38,2)(49,1)(49,8)(65,3)(76,5)(97,4)
堆排序后:(见图 9-10)
(13,6)(27,7)(38,2)(49,1)(49,8)(65,3)(76,5)(97,4)
```



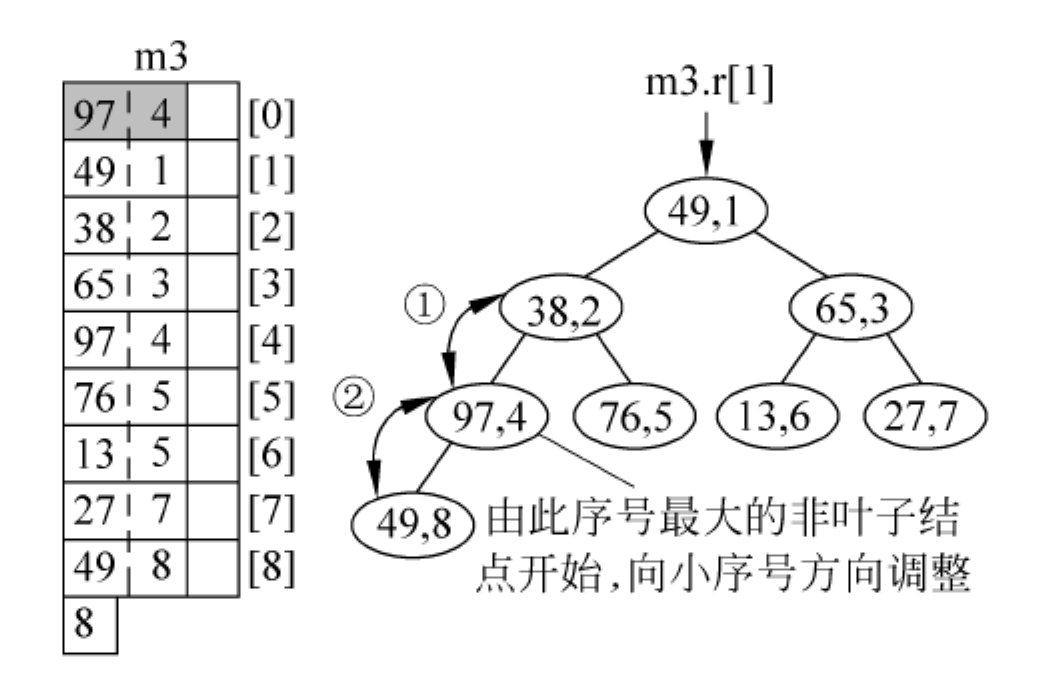

(a) 将m2.r赋给t的叶子结点

(b) 给t的非叶子结点赋值，t[0]已排序，赋给m2.r[0]

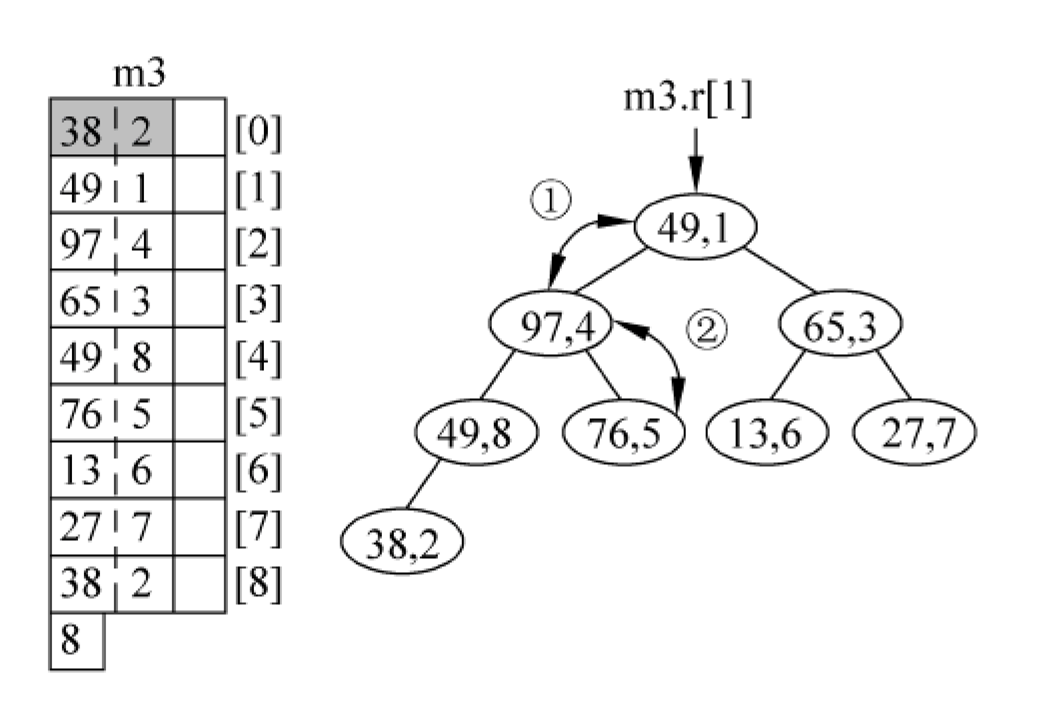
(c) 将叶子结点中已给m2.r赋过值的关键字改为∞，并沿根结点方向修改非叶子结点的值

图 9-9　树形选择排序

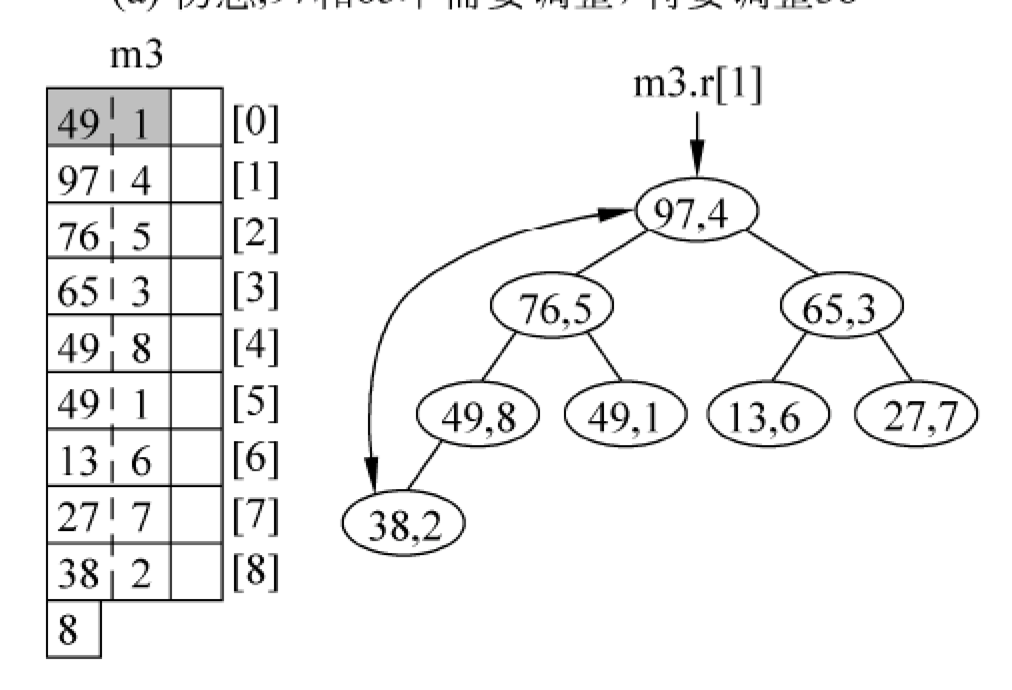
树形选择排序采用辅助数组 t 作为二叉树的顺序存储结构，将待排序的数据存于叶子结点去比较。它的优点是减少了比较次数；缺点是辅助数组 t 所占空间较大，接近数据个数的 2 倍。



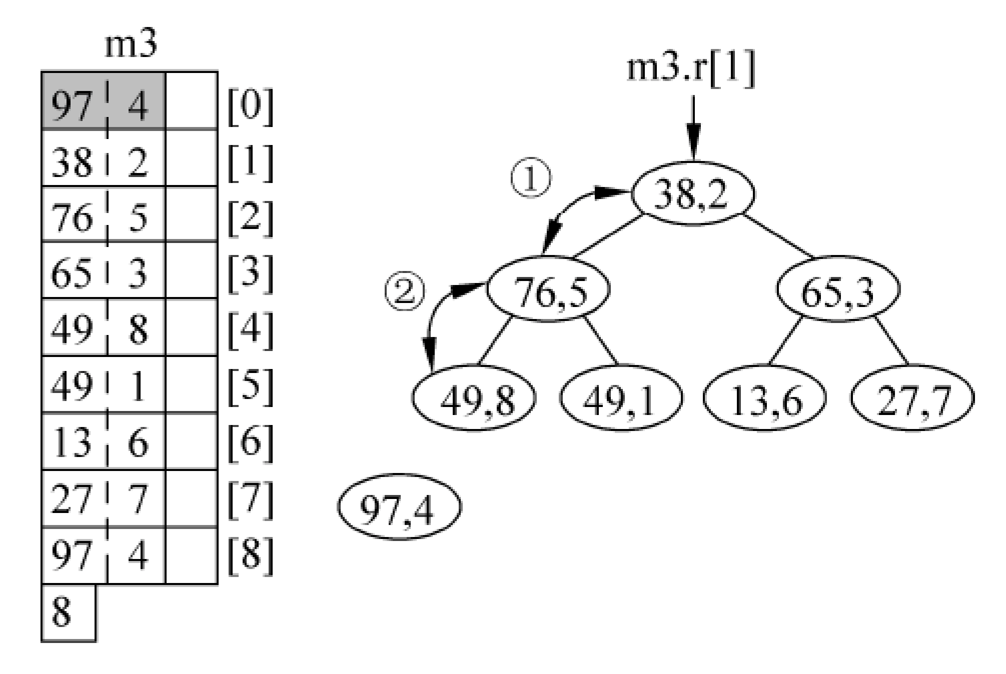

(a) 初态,97和65不需要调整，将要调整38



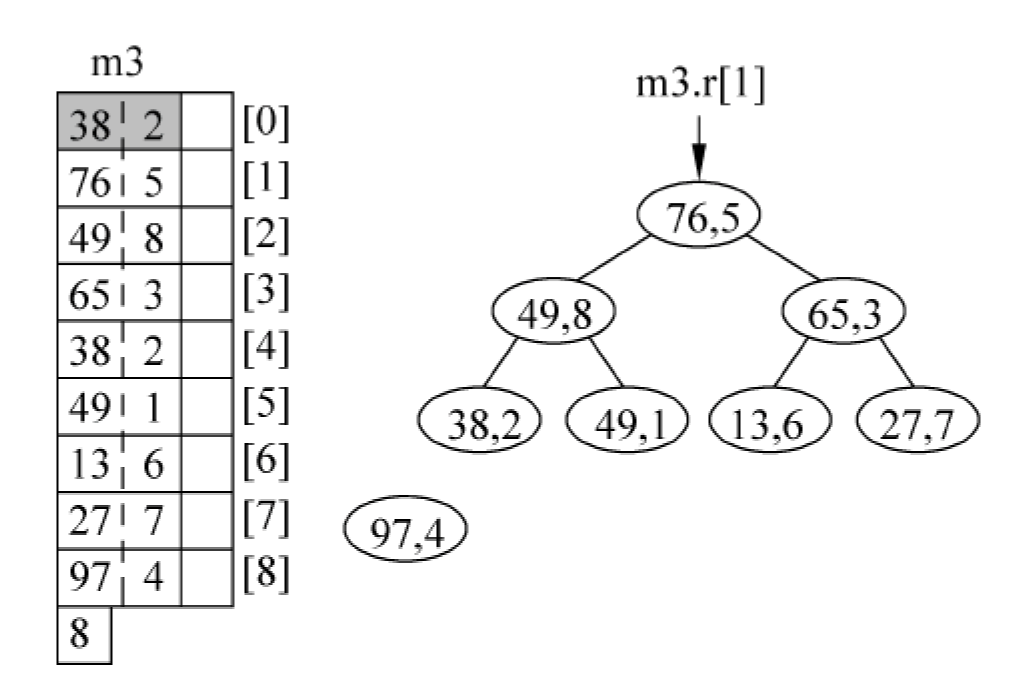

(b) 调整38后的结果，将要调整49



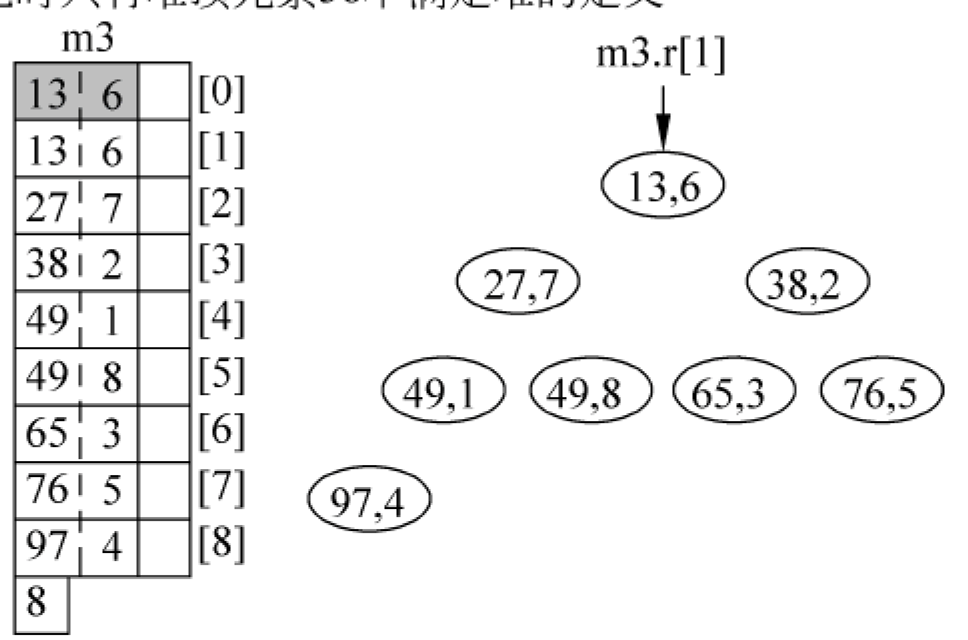

(c) 调整49后成为大顶堆，将交换97和38




(d) 大顶堆的顶元素沉底,并脱离二叉树(不再进入排序)，此时只有堆顶元素38不满足堆的定义




(e) 调整38后成为新的大顶堆




(f) 堆排序的最终结果

图 9-10　大顶堆(升序)排序示例

注：双向箭头表示将要进行的调整

堆排序是树形选择排序的改进,它不需要辅助数组。直接将顺序存储的数据看作一棵完全二叉树上的结点。这与 c6-1.h 定义的二叉树的顺序存储结构有相似之处,但堆排序的根结点存于[1]中([0]作临时存储之用)。所以,序号为 j 的结点,其双亲序号为$\lfloor j/2 \rfloor$;其左右孩子序号分别为 2j 和 2j+1。堆的定义是(以大顶堆(升序)为例):任何一个非叶子结点的关键字值都不小于它左右孩子结点的关键字值。所以,对于一个满足堆定义的顺序存储结构,虽然它的所有元素的排列并不是完全有序的,但堆顶元素[1]必定是值最大的元素。

堆排序分 3 步:首先,将完全无序的二叉树建为堆,其步骤是,对于所有非叶子结点,由序号最大的结点起,由下至上地将以该结点为根的子树调整为堆,如图 9-10(a)~(c)所示。然后,将堆顶元素和最后一个元素[i]交换,则最后一个元素[i]是排好序的,将此元素从堆中除去,即堆的规模-1。在尚未排好序的元素([1]~[i-1])中,除刚交换的堆顶元素外,

都符合堆的定义，如图 9-10(d)所示。此时，去掉最后一个结点的二叉树，其左右子树都已建堆，只须调整新的堆顶元素，使新的二叉树满足堆的定义，如图 9-10(e)所示。最后，所有元素都已排序，并脱离堆，堆为空，如图 9-10(f)所示。堆排序的优点是：堆顶元素顶多需要与其左右孩子之一(3 元素中的最大值)作交换。同时如果堆顶元素被交换，则只有交换的影响波及的小堆可能需要再作交换，且所有叶子结点(具有 n 个结点的完全二叉树只有⌊n/2⌋个非叶子结点)没有孩子，不用交换。这样就可用较少的比较、调整步骤，达到排序目的。

## 9.5 归并排序

数据文件 f9-3.txt 的内容如下：

```
7
49 1
38 2
65 3
97 4
76 5
13 6
27 7
```

```
// alg9-7.cpp 归并排序，包括算法 10.12～算法 10.14
#include"c1.h"
#include"c8-2.h" // 对两个数值型关键字比较的约定
typedef int KeyType; // 定义关键字的类型为整型
typedef int InfoType; // 定义其他数据项的类型为整型
#include"c9-1.h" // 记录的数据类型
#include"c9-2.h" // 顺序表类型的存储结构
#include"func9-1.cpp" // 配套的输入输出函数
void Merge(RedType SR[],RedType TR[],int i,int m,int n)
{ // 将有序的 SR[i..m]和 SR[m+1..n]归并为有序的 TR[i..n]。算法 10.12
  int j,k,p;
  for(j=m+1,k=i;i<=m&&j<=n;++k) // 将 SR 中记录由小到大地并入 TR
    if LQ(SR[i].key,SR[j].key)
      TR[k]=SR[i++];
    else
      TR[k]=SR[j++];
  if(i<=m)
    for(p=0;p<=m-i;p++)
      TR[k+p]=SR[i+p]; // 将剩余的 SR[i..m]复制到 TR
  if(j<=n)
    for(p=0;p<=n-j;p++)
      TR[k+p]=SR[j+p]; // 将剩余的 SR[j..n]复制到 TR
}
void MSort(RedType SR[],RedType TR1[],int s, int t)
```

```
{ // 将 SR[s..t]归并排序为 TR1[s..t]。算法 10.13
  int m;
  RedType TR2[MAX_SIZE + 1];
  if(s == t) // 只有 1 个元素待归并
    TR1[s] = SR[s]; // 直接赋值
  else // 有多个元素待归并
  { m = (s + t)/2; // 将 SR[s..t]平分为 SR[s..m]和 SR[m + 1..t]
    MSort(SR,TR2,s,m); // 递归地将 SR[s..m]归并为有序的 TR2[s..m]
    MSort(SR,TR2,m + 1,t); // 递归地将 SR[m + 1..t]归并为有序的 TR2[m + 1..t]
    Merge(TR2,TR1,s,m,t); // 将 TR2[s..m]和 TR2[m + 1..t]归并到 TR1[s..t]
  }
}
void MergeSort(SqList &L)
{ // 对顺序表 L 作归并排序。算法 10.14
  MSort(L.r,L.r,1,L.length);
  // 将顺序表 L.r[1..L.length]归并排序为有序的顺序表 L.r[1..L.length]
}
void main()
{
  FILE *f; // 文件指针类型
  SqList m; // 顺序表变量
  int i;
  f = fopen("f9-3.txt","r"); // 打开数据文件 f9-3.txt
  fscanf(f,"%d",&m.length); // 由数据文件输入数据元素个数给 m.length
  for(i = 1;i<= m.length;i++ ) // 给 m.r 赋值
    InputFromFile(f,m.r[i]); // 由数据文件输入数据元素的值并赋给 m1.r[i]
  fclose(f); // 关闭数据文件
  printf("排序前：\n");
  Print(m); // 输出排序前的顺序表 m
  MergeSort(m); // 对 m 调用归并排序法
  printf("排序后：\n");
  Print(m); // 输出排序后的顺序表 m
}
```

**程序运行结果**(以教科书中图 10.13 的数据为例)：

```
排序前：
(49,1)(38,2)(65,3)(97,4)(76,5)(13,6)(27,7)
排序后：
(13,6)(27,7)(38,2)(49,1)(65,3)(76,5)(97,4)
```

# 9.6 基数排序

```
// c9-4.h 基数排序的数据类型。在教科书第 286 页
#define MAX_NUM_OF_KEY 8 // 关键字项数的最大值
```

```
#define RADIX 10 // 关键字基数，此时是十进制整数的基数
#define MAX_SPACE 100
struct SLCell // 静态链表的结点类型(见图 9-11)
{ KeysType keys[MAX_NUM_OF_KEY]; // 关键字
  InfoType otheritems; // 其他数据项
  int next;
};
struct SLList // 静态链表类型(见图 9-12)
{ SLCell r[MAX_SPACE]; // 静态链表的可利用空间，r[0]为头结点
  int keynum; // 记录的当前关键字个数
  int recnum; // 静态链表的当前长度
};
typedef int ArrType[RADIX]; // 指针数组类型
```



图 9-11　静态链表的结点类型

数据文件 f9-4.txt 的内容如下：

```
10
278 1
109 2
063 3
930 4
589 5
184 6
505 7
269 8
008 9
083 10
```



图 9-12　静态链表存储结构

```
// alg9-8.cpp 链式基数排序，包括算法 10.15～算法 10.17
#include"c1.h"
typedef char KeysType; // 定义关键字类型为字符串型
typedef int InfoType; // 定义其他数据项为整型
#include"c9-4.h" // 基数排序的数据类型
typedef SLList SqList; // 定义 SqList 为 SLList 类型，以便利用 func9-2.cpp 中的函数
#define length recnum // 定义 length 为 recnum 类型，以便利用 func9-2.cpp 中的函数
#include"func9-2.cpp" // 算法 10.18
void MadeListFromFile(SLList &L,FILE* f)
{ // 通过文件 f 建立顺序表 L
  int i;
  fscanf(f,"%d",&L.recnum); // 由数据文件输入表长给 L.recnum
  for(i = 1;i<= L.recnum;i++ ) // 依次输入结点的值(除 next 域)
    fscanf(f,"%s%d",&L.r[i].keys,&L.r[i].otheritems);
  L.keynum = strlen(L.r[1].keys); // 将关键字的长度赋给 L.keynum(设关键字等长)
}
int ord(char c)
```

```
{ // 返回关键字 c 的序号
  return c - '0';
}
void Distribute(SLCell r[],int i,ArrType f,ArrType e)
{ // 静态链表 L 的 r 域中记录已按(keys[i - 1],…,keys[0])有序。本算法按第 i 个关键字
  // keys[i](keys[0]是最低位关键字)建立 RADIX 个子表,使同一子表中记录的 keys[i]相同。
  // f[0..RADIX - 1]和 e[0..RADIX - 1]分别指向各子表中第一个和最后一个记录。算法 10.15
  int j,p;
  for(j = 0;j<RADIX;++j)
    f[j] = 0; // 各子表初始化为空表
  for(p = r[0].next;p;p = r[p].next) // p 按链式结构依次指向静态链表的记录
  { j = ord(r[p].keys[i]); // 当前记录的第 i 位关键字的序号,以下将当前记录按序号插入子表
    if(!f[j]) // 子表[j]空
      f[j] = p; // 表头指向当前记录
    else // 子表[j]不空
      r[e[j]].next = p; // 修改原子表[j]的表尾记录的 next 域指向当前记录
    e[j] = p; // 设置表尾指针指向 p 所指的新表尾记录
  }
  printf("\nf[j] = "); // 以下输出表头指针 f[]和表尾指针 e[],新增
  for(j = 0;j<RADIX;++j)
    printf("%3d",f[j]);
  printf("\ne[j] = ");
  for(j = 0;j<RADIX;++j)
    if(f[j])
      printf("%3d",e[j]);
    else
      printf("%3d",0);
  printf("\n");
}
int succ(int i)
{ // 求后继函数
  return ++i;
}
void Collect(SLCell r[],ArrType f,ArrType e)
{ // 本算法按 keys[i]自小至大地将 f[0..RADIX - 1]所指各子表依次链接成一个链表,
  // e[0..RADIX - 1]为各子表的尾指针。算法 10.16
  int j,t;
  for(j = 0;!f[j];j = succ(j)); // 找第 1 个非空子表[j],succ 为求后继函数
  r[0].next = f[j]; // r[0].next 指向第 1 个非空子表[j]的第 1 个元素
  t = e[j]; // t 指向第 1 个非空子表[j]的表尾元素
  while(j<RADIX - 1) // 未到最后一位关键字
  { for(j = succ(j);j<RADIX - 1&&!f[j];j = succ(j)); // 找下一个非空子表
    if(f[j]) // 子表不空
    { r[t].next = f[j]; // 链接两个非空子表
```

```
        t = e[j]; // t指向新的表尾元素
      }
    }
    r[t].next = 0; // 表尾
}
void Print2(SLList L)
{ // 按数组序号输出静态链表
    int i = 0;
    printf("keynum = %d recnum = %d i = %d next = %d\n",L.keynum,L.recnum,i,
    L.r[i].next);
    for(i = 1;i<= L.recnum;i++ )
      printf("i = %d keys = %s otheritems = %d next = %d\n",i,L.r[i].keys,
      L.r[i].otheritems,L.r[i].next);
}
void PrintLL(SLList L)
{ // 按链表顺序输出静态链表L
    int i = L.r[0].next;
    while(i)
    { printf("%s",L.r[i].keys);
      i = L.r[i].next;
    }
}
void RadixSort(SLList &L)
{ // L是采用静态链表表示的顺序表。对L作基数排序,使得L成为按关键字
  // 自小到大的有序静态链表,L.r[0]为头结点。算法10.17
    int i,j = 1;
    ArrType f,e;
    for(i = 0;i<L.recnum;++i) // 将L改造为静态链表
      L.r[i].next = i + 1;
    L.r[L.recnum].next = 0;
    for(i = L.keynum - 1;i>= 0;--i,++j) // 按最低位优先依次对各关键字进行分配和收集,修改
    { Distribute(L.r,i,f,e); // 第i趟分配
      printf("第%d趟分配后:\n",j);
      Print2(L); // 按序号输出排序前的静态链表m
      Collect(L.r,f,e); // 第i趟收集
      printf("第%d趟收集后:\n",j);
      Print2(L); // 按序号输出静态链表L
      PrintLL(L); // 按链表顺序输出静态链表L
    }
}
void main()
{
    FILE *f; // 文件指针类型
    SLList m; // 静态链表变量
```

```
    int * adr;
    f = fopen("f9-4.txt","r"); // 打开数据文件 f9-4.txt
    MadeListFromFile(m,f); // 通过文件 f 建立静态链表 m
    fclose(f); // 关闭数据文件
    printf("排序前(next 域还未赋值):\n");
    Print2(m); // 按序号输出排序前的静态链表 m
    RadixSort(m); // 对 m 调用基数排序法,使得 m 成为有序静态链表
    adr = (int * )malloc((m.recnum + 1) * sizeof(int)); // 动态生成 adr 数组
    Sort(m,adr); // 求得 adr[1..m.length],adr[i]为静态链表 m 的第 i 个最小记录的序号
    Rearrange(m,adr); // 按 adr[]重排 m.r,使其成为有序的顺序表
    free(adr); // 释放 adr 所指的存储空间
    printf("\n 重排记录后(next 域不起作用):\n");
    Print2(m); // 按序号输出重新排序后的静态链表 m
}
```

**程序运行结果**(以教科书中图 10.14 的数据为例):

```
排序前(next 域还未赋值):(见图 9-13)
keynum = 3 recnum = 10 i = 0 next = 20480
i = 1 keys = 278 otheritems = 1 next = 25956
i = 2 keys = 109 otheritems = 2 next = 28532
i = 3 keys = 063 otheritems = 3 next = 25455
i = 4 keys = 930 otheritems = 4 next = 26144
i = 5 keys = 589 otheritems = 5 next = 8293
i = 6 keys = 184 otheritems = 6 next = 28448
i = 7 keys = 505 otheritems = 7 next = 26992
i = 8 keys = 269 otheritems = 8 next = 25458
i = 9 keys = 008 otheritems = 9 next = 8303
i = 10 keys = 083 otheritems = 10 next = 25960
```

m

| | | | |
|---|---|---|---|
| | | | [0] |
| 278 | 1 | | [1] |
| 109 | 2 | | [2] |
| 063 | 3 | | [3] |
| 930 | 4 | | [4] |
| 589 | 5 | | [5] |
| 184 | 6 | | [6] |
| 505 | 7 | | [7] |
| 269 | 8 | | [8] |
| 008 | 9 | | [9] |
| 083 | 10 | | [10] |
| | ⋮ | | ⋮ |
| | | | [99] |
| 3 | | | |
| 10 | | | |

图 9-13　排序前

```
f[j] = 4 0 0  3 6 7 0 0 1 2
e[j] = 4 0 0 10 6 7 0 0 9 8
第 1 趟分配后:(见图 9-14)
keynum = 3 recnum = 10 i = 0 next = 1
i = 1 keys = 278 otheritems = 1 next = 9
i = 2 keys = 109 otheritems = 2 next = 5
i = 3 keys = 063 otheritems = 3 next = 10
i = 4 keys = 930 otheritems = 4 next = 5
i = 5 keys = 589 otheritems = 5 next = 8
i = 6 keys = 184 otheritems = 6 next = 7
i = 7 keys = 505 otheritems = 7 next = 8
i = 8 keys = 269 otheritems = 8 next = 9
i = 9 keys = 008 otheritems = 9 next = 10
i = 10 keys = 083 otheritems = 10 next = 0
```

m

| | | | |
|---|---|---|---|
| | | 1 | [0] |
| 278 | 1 | 9 | [1] |
| 109 | 2 | 5 | [2] |
| 063 | 3 | 10 | [3] |
| 930 | 4 | 5 | [4] |
| 589 | 5 | 8 | [5] |
| 184 | 6 | 7 | [6] |
| 505 | 7 | 8 | [7] |
| 269 | 8 | 9 | [8] |
| 008 | 9 | 10 | [9] |
| 083 | 10 | 0 | [10] |
| | ⋮ | | ⋮ |
| | | | [99] |
| 3 | | | |
| 10 | | | |

图 9-14　第 1 趟分配后

第 1 趟收集后：（见图 9-15）

```
keynum = 3 recnum = 10 i = 0 next = 4
i = 1 keys = 278 otheritems = 1 next = 9
i = 2 keys = 109 otheritems = 2 next = 5
i = 3 keys = 063 otheritems = 3 next = 10
i = 4 keys = 930 otheritems = 4 next = 3
i = 5 keys = 589 otheritems = 5 next = 8
i = 6 keys = 184 otheritems = 6 next = 7
i = 7 keys = 505 otheritems = 7 next = 1
i = 8 keys = 269 otheritems = 8 next = 0
i = 9 keys = 008 otheritems = 9 next = 2
i = 10 keys = 083 otheritems = 10 next = 6
930 063 083 184 505 278 008 109 589 269
f[j]=  7 0 0 4 0 0 3 1 10 0
e[j]=  2 0 0 4 0 0 8 1  5 0
```

第 2 趟分配后：（见图 9-16）

```
keynum = 3 recnum = 10 i = 0 next = 4
i = 1 keys = 278 otheritems = 1 next = 9
i = 2 keys = 109 otheritems = 2 next = 5
i = 3 keys = 063 otheritems = 3 next = 8
i = 4 keys = 930 otheritems = 4 next = 3
i = 5 keys = 589 otheritems = 5 next = 8
i = 6 keys = 184 otheritems = 6 next = 5
i = 7 keys = 505 otheritems = 7 next = 9
i = 8 keys = 269 otheritems = 8 next = 0
i = 9 keys = 008 otheritems = 9 next = 2
i = 10 keys = 083 otheritems = 10 next = 6
```

第 2 趟收集后：（见图 9-17）

```
keynum = 3 recnum = 10 i = 0 next = 7
i = 1 keys = 278 otheritems = 1 next = 10
i = 2 keys = 109 otheritems = 2 next = 4
i = 3 keys = 063 otheritems = 3 next = 8
i = 4 keys = 930 otheritems = 4 next = 3
i = 5 keys = 589 otheritems = 5 next = 0
i = 6 keys = 184 otheritems = 6 next = 5
i = 7 keys = 505 otheritems = 7 next = 9
i = 8 keys = 269 otheritems = 8 next = 1
i = 9 keys = 008 otheritems = 9 next = 2
i = 10 keys = 083 otheritems = 10 next = 6
505 008 109 930 063 269 278 083 184 589
f[j]=   9 2 8 0 0 7 0 0 0 4
e[j]=  10 6 1 0 0 5 0 0 0 4
```

| | m | | |
|---|---|---|---|
| | | 4 | [0] |
| 278 | 1 | 9 | [1] |
| 109 | 2 | 5 | [2] |
| 063 | 3 | 10 | [3] |
| 930 | 4 | 3 | [4] |
| 589 | 5 | 8 | [5] |
| 184 | 6 | 7 | [6] |
| 505 | 7 | 1 | [7] |
| 269 | 8 | 0 | [8] |
| 008 | 9 | 2 | [9] |
| 083 | 10 | 6 | [10] |
| | ⋮ | | ⋮ |
| | | | [99] |
| 3 | | | |
| 10 | | | |

图 9-15　第 1 趟收集后

| | m | | |
|---|---|---|---|
| | | 4 | [0] |
| 278 | 1 | 9 | [1] |
| 109 | 2 | 5 | [2] |
| 063 | 3 | 8 | [3] |
| 930 | 4 | 3 | [4] |
| 589 | 5 | 8 | [5] |
| 184 | 6 | 5 | [6] |
| 505 | 7 | 9 | [7] |
| 269 | 8 | 0 | [8] |
| 008 | 9 | 2 | [9] |
| 083 | 10 | 6 | [10] |
| | ⋮ | | ⋮ |
| | | | [99] |
| 3 | | | |
| 10 | | | |

图 9-16　第 2 趟分配后

| | m | | |
|---|---|---|---|
| | | 7 | [0] |
| 278 | 1 | 10 | [1] |
| 109 | 2 | 4 | [2] |
| 063 | 3 | 8 | [3] |
| 930 | 4 | 3 | [4] |
| 589 | 5 | 0 | [5] |
| 184 | 6 | 5 | [6] |
| 505 | 7 | 9 | [7] |
| 269 | 8 | 1 | [8] |
| 008 | 9 | 2 | [9] |
| 083 | 10 | 6 | [10] |
| | ⋮ | | ⋮ |
| | | | [99] |
| 3 | | | |
| 10 | | | |

图 9-17　第 2 趟收集后

第 3 趟分配后：(见图 9-18)
keynum = 3 recnum = 10 i = 0 next = 7
i = 1 keys = 278 otheritems = 1 next = 10
i = 2 keys = 109 otheritems = 2 next = 6
i = 3 keys = 063 otheritems = 3 next = 10
i = 4 keys = 930 otheritems = 4 next = 3
i = 5 keys = 589 otheritems = 5 next = 0
i = 6 keys = 184 otheritems = 6 next = 5
i = 7 keys = 505 otheritems = 7 next = 5
i = 8 keys = 269 otheritems = 8 next = 1
i = 9 keys = 008 otheritems = 9 next = 3
i = 10 keys = 083 otheritems = 10 next = 6

| | m | | |
|---|---|---|---|
| | | 7 | [0] |
| 278 | 1 | 10 | [1] |
| 109 | 2 | 6 | [2] |
| 063 | 3 | 10 | [3] |
| 930 | 4 | 3 | [4] |
| 589 | 5 | 0 | [5] |
| 184 | 6 | 5 | [6] |
| 505 | 7 | 5 | [7] |
| 269 | 8 | 1 | [8] |
| 008 | 9 | 3 | [9] |
| 083 | 10 | 6 | [10] |
| | ⋮ | | ⋮ |
| | | | [99] |
| 3 | | | |
| 10 | | | |

图 9-18 第 3 趟分配后

第 3 趟收集后：(见图 9-19)
keynum = 3 recnum = 10 i = 0 next = 9
i = 1 keys = 278 otheritems = 1 next = 7
i = 2 keys = 109 otheritems = 2 next = 6
i = 3 keys = 063 otheritems = 3 next = 10
i = 4 keys = 930 otheritems = 4 next = 0
i = 5 keys = 589 otheritems = 5 next = 4
i = 6 keys = 184 otheritems = 6 next = 8
i = 7 keys = 505 otheritems = 7 next = 5
i = 8 keys = 269 otheritems = 8 next = 1
i = 9 keys = 008 otheritems = 9 next = 3
i = 10 keys = 083 otheritems = 10 next = 2
008 063 083 109 184 269 278 505 589 930

| | m | | |
|---|---|---|---|
| | | 9 | [0] |
| 278 | 1 | 7 | [1] |
| 109 | 2 | 6 | [2] |
| 063 | 3 | 10 | [3] |
| 930 | 4 | 0 | [4] |
| 589 | 5 | 4 | [5] |
| 184 | 6 | 8 | [6] |
| 505 | 7 | 5 | [7] |
| 269 | 8 | 1 | [8] |
| 008 | 9 | 3 | [9] |
| 083 | 10 | 2 | [10] |
| | ⋮ | | ⋮ |
| | | | [99] |
| 3 | | | |
| 10 | | | |

图 9-19 第 3 趟收集后

重排记录后(next 域不起作用)：(见图 9-20)
keynum = 3 recnum = 10 i = 0 next = 6
i = 1 keys = 008 otheritems = 9 next = 3
i = 2 keys = 063 otheritems = 3 next = 10
i = 3 keys = 083 otheritems = 10 next = 2
i = 4 keys = 109 otheritems = 2 next = 6
i = 5 keys = 184 otheritems = 6 next = 8
i = 6 keys = 269 otheritems = 8 next = 1
i = 7 keys = 278 otheritems = 1 next = 7
i = 8 keys = 505 otheritems = 7 next = 5
i = 9 keys = 589 otheritems = 5 next = 4
i = 10 keys = 930 otheritems = 4 next = 0

| | m | | |
|---|---|---|---|
| | | | [0] |
| 008 | 9 | | [1] |
| 063 | 3 | | [2] |
| 083 | 10 | | [3] |
| 109 | 2 | | [4] |
| 184 | 6 | | [5] |
| 269 | 8 | | [6] |
| 278 | 1 | | [7] |
| 505 | 7 | | [8] |
| 589 | 5 | | [9] |
| 930 | 4 | | [10] |
| | ⋮ | | ⋮ |
| | | | [99] |
| 3 | | | |
| 10 | | | |

图 9-20 重排记录后

## 9.7 各种内部排序方法的比较讨论

算法 10.18 在 func9-2.cpp 中。

# 第10章

# 外 部 排 序

第 9 章介绍的内部排序需要把待排序的数据先全部放在内存中，然后再排序，这就限制了待排序数据的规模。当数据量特别大时，软件的数据区有可能放不下。algo10-1. cpp 可说明这个问题。

```
// algo10-1.cpp
#define N 32767
void main()
{
int a[N];
}
```

在 Borland C++ 3.1 编译软件下，当 N≤32 767 时，编译 algo10-1. cpp 顺利通过。而当 N>32 767 时，编译 algo10-1. cpp 时出错："Array size too large"（数组太大）。这个例子说明，要对多于 32 767 个整数进行内部排序是不可能的。首先就没有足够的内存空间存放这些数据。在 Visual C++ 6.0 环境下，N 的范围可大一些，但也是有限的（在 Visual C++ 6.0 环境下，增加了语句：for(int i=0;i<N;i++) a[i]=i; ）。当 N≤259 946 时，可运行 algo10-1. cpp，而当 N>259 946 时，运行 algo10-1. cpp 会出现消息框，提示：该程序执行了非法操作。经 Debug 查看，原因是："Stack Overflow"（堆栈溢出）。

本章介绍在没有足够的内存空间存放待排序数据的情况下，对数据进行排序的方法。

## 10.1 外部排序的方法

当存放待排序记录的文件所占存储空间大于可用的内存空间时（把这样的文件称为"大文件"），显然不能采用第 9 章介绍的内部排序的方法，将文件中的所有记录一次读入内存进行排序。在这种情况下，一般是根据可用内存的大小，先把大文件分几次读入内存，对每次读入的记录进行内部排序，生成几个临时的有序子文件，如图 10-1 所示。再将这几个临时的有序子文件归并成一个有序的大文件，这个过程就称为"外部排序"。

假设内存空间只能存放 3 个记录，则有 15 个记录的文件就是"大文件"了。无序的大数据文件 f10-1. txt 的内容如下：

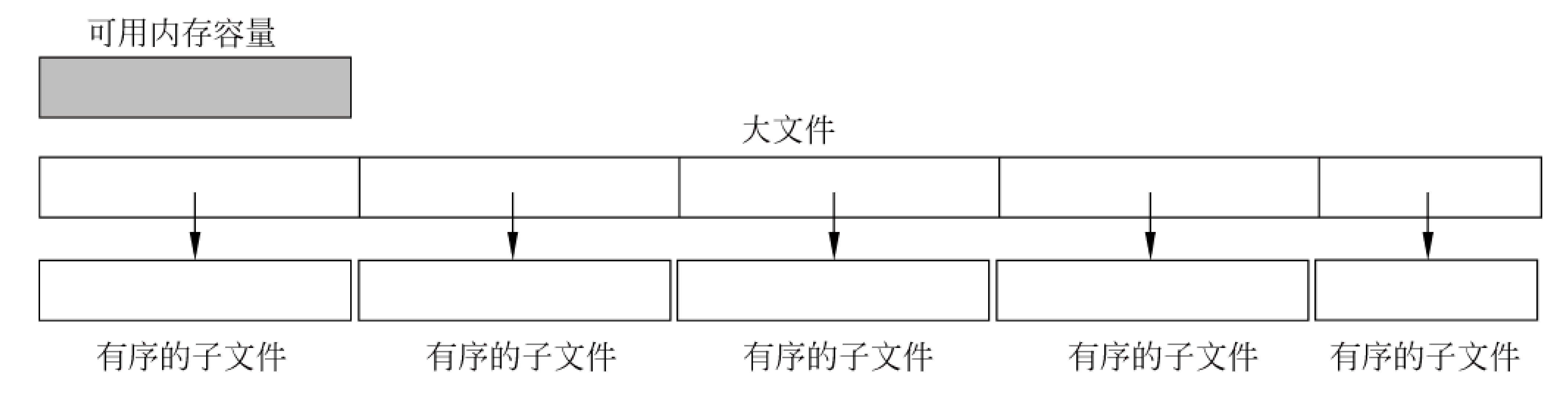



图 10-1　将 1 个无序的大文件通过内部排序生成几个有序的子文件

```
16 1
15 2
10 3
20 4
9 5
18 6
22 7
20 8
40 9
15 10
25 11
6 12
12 13
48 14
37 15
```

```
// algo10-2.cpp 将无序的大文件 f10-1.txt(记录个数 = k * N)分成 k 个长度为 N 的有序小文件的程序
#include"c1.h"
#include"c8-2.h" // 对两个数值型关键字比较的约定
typedef int KeyType; // 定义关键字的类型为整型
typedef int InfoType; // 定义其他数据项的类型为整型
#include"c9-1.h" // 记录的数据类型
#include"c9-2.h" // 顺序表类型的存储结构
#include"bo9-1.cpp" // 顺序表插入排序的函数
#include"func9-1.cpp" // 配套的输入输出函数
#define k 5 // k 路归并
#define N 3 // 设每个小文件最多有 N 个数据(可将整个文件一次读入内存的称为小文件)
void main()
{
  SqList m; // 顺序表,用于对小文件进行内部排序
  FILE *f,*g; // 文件指针
  char filename[3]; // 有序小文件名
  int i,j;
  f = fopen("f10-1.txt","r"); // 以读的方式打开含有 k * N 个记录的未排序大文件 f10-1.txt
  for(i = 0;i<k;i++) // 将大文件 f10-1.txt 的数据分成 k 组
  { for(j = 1;j<= N;j++) // 每组 N 个数据
```

```
    InputFromFile(f,m.r[j]); // 由数据文件输入值并赋给 m.r[j]
  m.length = N; // 顺序表的记录长度
  BInsertSort(m); // 对顺序表 m 作折半插入排序,使 m.r 成为有序表
  itoa(i,filename,10); // i 作为临时的有序小文件的文件名
  g = fopen(filename,"w"); // 依次以写的方式打开文件 0,1,…,k-1
  printf("有序子文件 %s 的数据为:",filename);
  Print(m); // 输出排序后的 m
  for(j = 1;j<= N;j++ ) // 依次将已排序的 m.r[j]写入小文件 0,1,…,k-1
    fprintf(g,"%d %d\n",m.r[j].key,m.r[j].otherinfo);
  fclose(g); // 依次关闭小文件 0,1,…,k-1
 }
 fclose(f); // 关闭未排序的大文件 f10-1.txt
}
```

**程序运行结果:**

```
有序子文件 0 的数据为:(10,3)(15,2)(16,1)
有序子文件 1 的数据为:(9,5)(18,6)(20,4)
有序子文件 2 的数据为:(20,8)(22,7)(40,9)
有序子文件 3 的数据为:(6,12)(15,10)(25,11)
有序子文件 4 的数据为:(12,13)(37,15)(48,14)
```

运行 algo10-2.cpp,依次读取无序的大文件 f10-1.txt 中的数据,生成了 5 个文件名分别为 0、1、2、3、4 的有序子文件,它们的内容如下:

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 10 3 | 9 5 | 20 8 | 6 12 | 12 13 |
| 15 2 | 18 6 | 22 7 | 15 10 | 37 15 |
| 16 1 | 20 4 | 40 9 | 25 11 | 48 14 |

这 5 个有序子文件可供下一节的程序 algo10-3.cpp 调用归并成一个有序的大文件。

## 10.2 多路平衡归并的实现

将多个有序的子文件归并成 1 个有序的大文件时,每归并 1 次,就要在外存读 1 次记录。如果减少归并次数,就可减少读文件的次数。但减少归并次数,就要多路归并(同时将几个文件进行归并)。而多路归并要在多个关键字中比较,找到最小值,显然是较慢的。利用“败者树”可减少比较次数。

“败者树”是完全二叉树,所以采用顺序存储结构。但它和第 6 章介绍的二叉树顺序存储结构又有两点不同。

(1) 根结点在[1]中([0]存胜出者)。所以,序号为 j 的结点,其双亲结点的序号为$\lfloor j/2 \rfloor$。

(2) “败者树”是由 2 种不同类型的结点组成的:

① **叶子结点**:是外部结点,其类型为记录类型。每个叶子结点中的值为从相应的有序子文件读出的当前记录。叶子结点的个数恰为待归并的文件数 k;

② **非叶子结点**：是度为 2 的内部结点，其类型为整型，存储叶子结点的序号。非叶子结点的个数＝叶子结点的个数 k－1。

序号为 s 的叶子结点，其双亲结点的序号为$\lfloor (k+s)/2 \rfloor$（叶子结点的序号由[0]起）。每个非叶子结点中的值为其左右 2 棵子树中的 2 个胜者再相比时，其中“败者”的序号，“胜者”再向上一级去比较。

最终“胜者”结点的序号存于败者树的根结点之上的[0]，再存于大文件中。“胜者”原先所在的有序子文件的下一个数据取代“胜者”在叶子结点中的位置，继续比较，求出新的“胜者”。但在继续比较时，它不必和所有叶子的关键字去比较，只须沿着从相应的叶子结点到根结点的路径去调整“败者树”。这就大大减少了比较次数。

一个新的叶子结点取代“胜者”加入到“败者树”中，形成新的“败者树”，即求出新的最小值，需要进行比较的次数＝这个结点所在的层数－1。

具有 N 个叶子结点的“败者树”，共有 2N－1 个结点。其深度为$\lfloor \log_2(2N-1) \rfloor+1$，求出最小值需要进行比较的次数＝$\lfloor \log_2(2N-1) \rfloor$。而用一般的方法，需要比较 N－1 次。设 N＝128，利用“败者树”只需比较 7 次，否则需比较 127 次。

bo10-1.cpp 是 k-路平衡归并要调用的函数。归并路数 k 的值在调用 bo10-1.cpp 的主程序中确定。

algo10-3.cpp 是利用“败者树”进行外部排序的完整程序。同时打开在上一节运行 algo10-2.cpp 产生的 5 个含有 3 个记录、并按关键字有序的子文件 0、1、2、3、4（其关键字正是教科书中图 11.4 的数据），利用“败者树”归并成 1 个有序的大文件 f10-3.txt。

```
// bo10-1.cpp k-路平衡归并的函数，包括算法 11.1～算法 11.3
void input(int i,RedType &a)
{ // 从第 i 个文件(归并段)读入记录到 a
  fscanf(fp[i],"%d %d",&a.key,&a.otherinfo);
}
void output(RedType a)
{ // 将 a 写至全局变量 fp[k]所指的文件中并输出
  static int col = 0; // 静态变量，计数，换行用
  fprintf(fp[k],"%d %d\n",a.key,a.otherinfo); // 将记录 a 写入大文件 fp[k]
  Print(a); // 输出 a，新增
  if( ++col % M == 0)
    printf("\n"); // 换行
}
void Adjust(LoserTree ls,int s) // 算法 11.2
{ // 沿从叶子结点全局变量 b[s]到根结点全局变量 ls[1]的路径调整败者树，胜者存 ls[0]
  int i,t;
  t = (s + k)/2; // t 是全局变量 b[s]的双亲结点的序号
  while(t>0) // t 还在败者树上
  { if(b[s].key>b[ls[t]].key) // b[s]的关键字大于其双亲结点的关键字(是败者)
    { i = s; // 交换 s 和 ls[t]
      s = ls[t]; // s 指示胜者，将和 b[s]的双亲结点的双亲结点比较
      ls[t] = i; // b[s]的双亲结点指示败者 b[s]
    }
    t = t/2; // t 为 b[s]的双亲结点的双亲结点的序号
```

```
  }
  ls[0] = s; // 胜者存于败者树之外的[0]
}
void CreateLoserTree(LoserTree ls)
{ // 已知全局变量 b[0]～b[k-1]为完全二叉树全局变量 ls 的叶子结点，存有 k 个关键字，
  // 沿从序号最大的叶子到根的 k 条路径将 ls 调整成为败者树。算法 11.3
  int i;
  b[k].key = MIN_KEY; // [k]中存最小关键字
  for(i = 1;i<k;++i)
    ls[i] = k; // 设置 ls 中"败者"的初值为有最小关键字的序号(调整中必会被败者取代)
  for(i = k-1;i>= 0;-- i) // 依次从序号最大的叶子结点 b[k-1]～b[0]出发调整败者树
    Adjust(ls,i); // 沿从叶子结点 b[i]到根结点 ls[1]的路径调整败者树，胜者存 ls[0]
}
void K_Merge(LoserTree ls)
{ // 利用全局变量败者树 ls 将编号从[0]～[k-1]的 k 个输入归并段中的记录归并到输出归并段。
  // 全局变量 b[0]～b[k-1]为败者树上的 k 个叶子结点，分别存放 k 个输入归并段中当前
  // 记录的关键字。修改算法 11.1
  int i;
  for(i = 0;i<k;++ i) // 依次从 k 个输入归并段
  { input(i,b[i]); // 读入该段第 1 个记录到外结点 b[i]
    if(feof(fp[i])) // 读记录失败(文件中无记录)
      b[i].key = MAX_KEY; // 设置外结点关键字的值为最大
  }
  CreateLoserTree(ls);
  // 初建败者树 ls[1..k-1]，树外结点 ls[0]指示 b[0]～b[k-1]中关键字最小者(胜者)的序号
  while(b[ls[0]].key!= MAX_KEY) // 胜出的关键字不是最大关键字(b[ls[0]]是文件中的记录)
  { output(b[ls[0]]); // 将 b[ls[0]]输出到文件
    input(ls[0],b[ls[0]]); // 从编号为 ls[0]的输入归并段中读入下一个记录到 b[ls[0]]
    if(feof(fp[ls[0]])) // 读记录失败(文件中已无记录)
      b[ls[0]].key = MAX_KEY; // 设置外结点关键字的值为最大
    Adjust(ls,ls[0]);
    // 沿从取得新值的叶子结点 b[ls[0]]到根结点 ls[1]的路径调整败者树，选出新的最小关键字
  }
}
```

```
// algo10-3.cpp 调用 bo10-1.cpp 的程序(运行 algo10-2.cpp 后运行此程序)
#include"c1.h"
typedef int KeyType; // 定义关键字的类型为整型
typedef int InfoType; // 定义其他数据项的类型为整型
#include"c9-1.h" // 记录的数据类型
#define k 5 // k(5)路归并，以下 2 行取 1 行。第 6 行
//#define k 3 // 3 路归并。第 7 行
FILE *fp[k+1]; // k+1 个文件指针(fp[k]为大文件指针)，全局变量
typedef int LoserTree[k];
// [1..k-1]是败者树的非叶子结点，[0]中是胜者，存相应叶子的序号
RedType b[k+1]; // [0..k-1]是败者树的叶子(外结点)，[k]存最小关键字
#define MIN_KEY INT_MIN // 最小关键字
```

```
#define MAX_KEY INT_MAX // 最大关键字
#define M 10 // 设输出 M 个排序后的数据换行
void Print(RedType t)
{ printf("(%d,%d)",t.key,t.otherinfo);
}
#include"bo10-1.cpp" // k 路平衡归并的函数
void main()
{
  LoserTree ls; // 败者树,长度为 k 的整型数组
  int i;
  char outfile[10],filename[3]; // 大文件名,有序小文件名
  for(i=0;i<k;i++)
  { itoa(i,filename,10); // i 作为临时的有序小文件的文件名
    fp[i]=fopen(filename,"r"); // 以读的方式打开已排序的小文件 0,1,…,k-1
  }
  printf("请输入排序后的大文件名:");
  scanf("%s",outfile); // 输入排序后的大文件名给 outfile
  fp[k]=fopen(outfile,"w"); // 以写的方式打开排序后的大文件 outfile
  printf("有序大文件 %s 的记录为:\n",outfile);
  K_Merge(ls); // 利用败者树将 k 个有序小文件中的记录归并为 1 个有序大文件
  printf("\n"); // 换行
  for(i=0;i<=k;i++)
    fclose(fp[i]); // 关闭文件 0,1,…,已排序的大文件 outfile
}
```

**程序运行结果**(以教科书中图 11.4 的数据为例,见图 10-2 和图 10-3):

```
请输入排序后的大文件名:f10-3.txt↙
有序大文件 f10-3.txt 的记录为:
(6,12)(9,5)(10,3)(12,13)(15,10)(15,2)(16,1)(18,6)(20,4)(20,8)
(22,7)(25,11)(37,15)(40,9)(48,14)
```

运行 algo10-3.cpp 产生的有序的大文件 f10-3.txt 的内容如下:

```
6 12
9 5
10 3
12 13
15 10
15 2
16 1
18 6
20 4
20 8
22 7
25 11
37 15
40 9
48 14
```

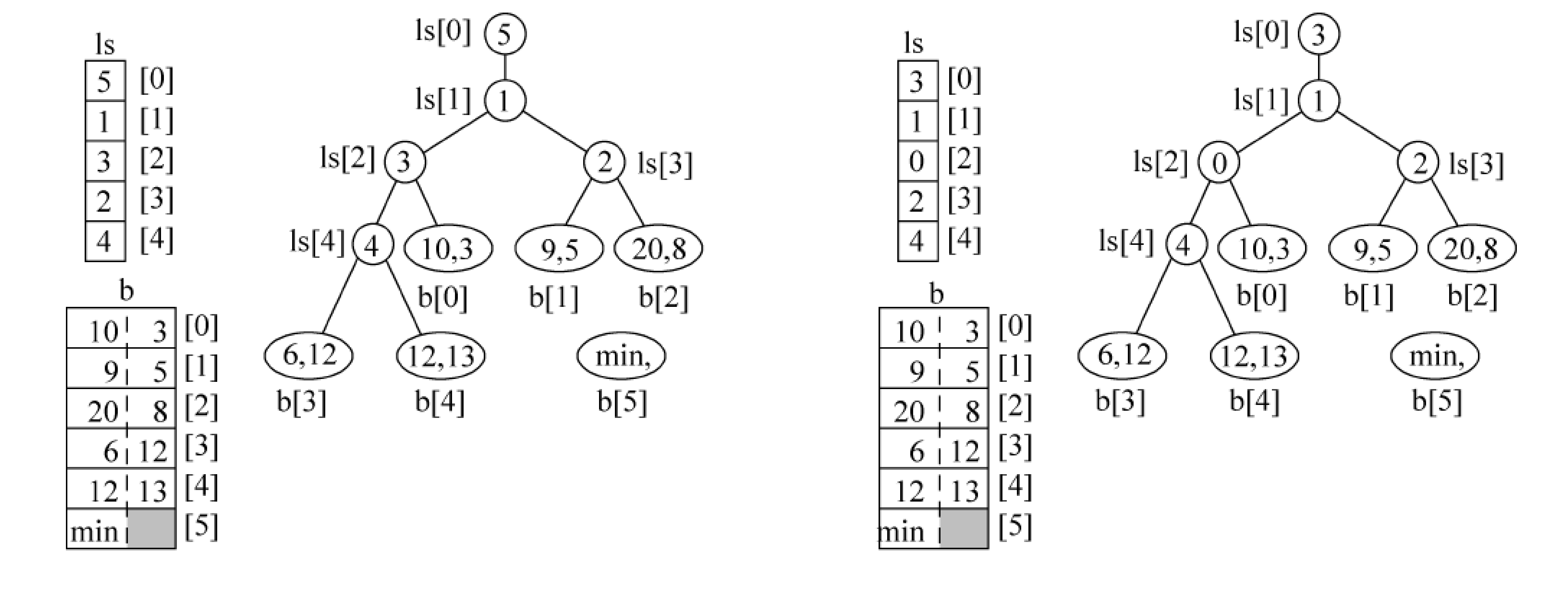
由此结点开始,向小序号方向调整

(a) 初态,在K_Merge()中给b和ls赋值

(b) 根据b[4]调整败者树

(c) 根据b[3]调整败者树

(d) 根据b[2]调整败者树




(e) 根据b[1]调整败者树

(f) 根据b[0]调整败者树,胜出者的序号在ls[0]

图 10-2 调用 CreateLoserTree()示例

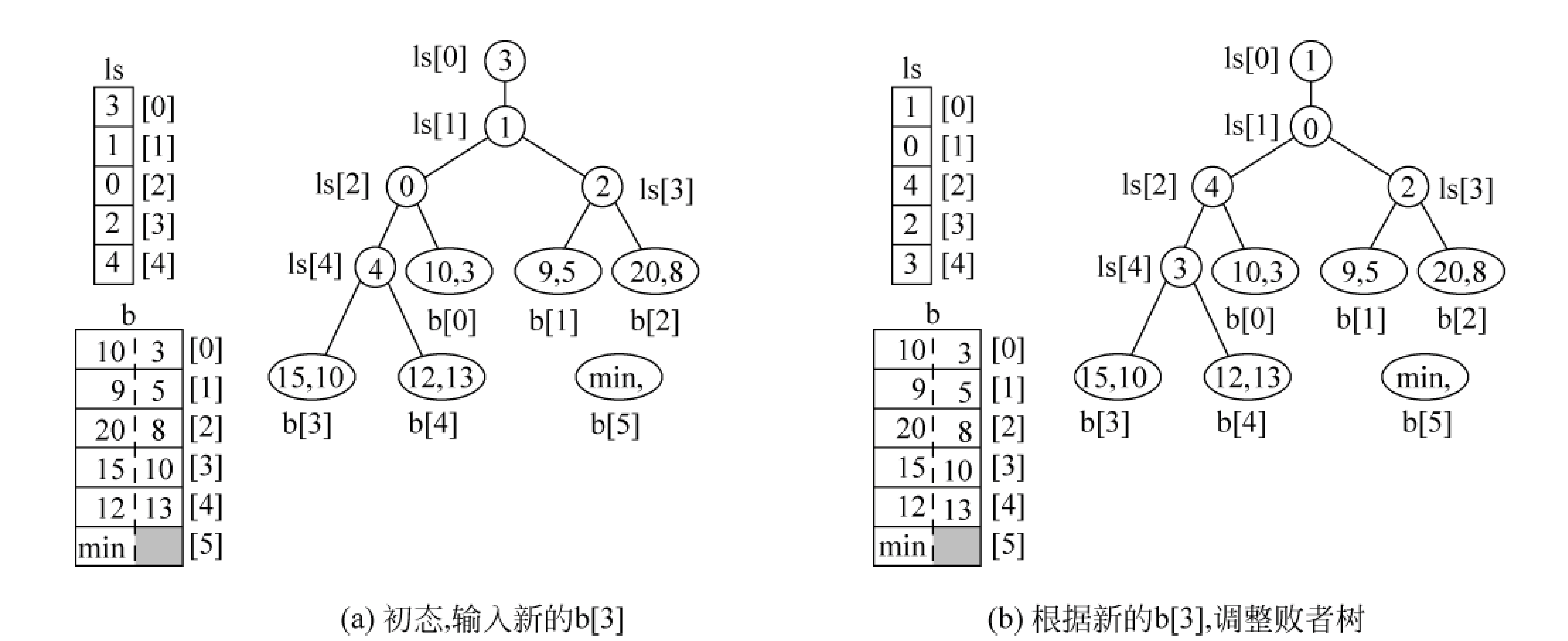


图 10-3 输入新的 b[3]后调用 Adjust()示例

## 10.3 置换-选择排序

要提高外部排序的效率,除了选择“败者树”归并方法外,还应该尽量使有序子文件较长,从而减少待归并文件的数量。但像图 10-1 所示那样由外存整体读入数据到内存,经内部排序后,再整体输出到外存形成的有序子文件,其长度不会大于可用内存容量。

置换-选择排序采取增多待排序记录个数的方法使生成的有序子文件更长。具体算法是:逐个把已排序的记录送入外存,再由“大文件”读入一个记录到内存中空出的位置,只要这个记录的关键字不小于刚送到外存的记录的关键字,该记录就会排到当前有序子文件中。

置换-选择排序在搜索内存中的最小值时,也是采用败者树。和多路平衡归并的败者树不同的是,外结点多了 1 个存储记录段号的 rnum 域。如果新读入的记录的关键字小于刚送到外存的记录的关键字,则设置新读入记录的段号比刚送到外存的记录的段号大 1,标志新读入的记录不排在当前的有序子文件中。下面以教科书中图 11.5 为例来说明:

设内存最多只能存放 6 个记录。首先由大文件 FI 中读取前 6 个记录(51,49,39,46,38,29)到内存 WA,它们都可以排到第 1 个有序小文件中,故设置其段号为 1。找出其中的最小值(29),存于外存文件 FO 中作为第 1 个记录。再由大文件 FI 读取第 7 个记录(14)到内存 WA,占据原 29 的位置。由于 14 小于 29,所以 14 不可能被排到第 1 个有序小文件中,故设置其段号为 2(可排到第 2 个有序小文件中)。再在内存 WA 中不小于 29 的(也就是段号为 1 的)关键字中找最小值,找到 38。将关键字为 38 的记录存于外存文件 FO 中作为第 2 个记录。再由大文件 FI 读取第 8 个记录(61)到内存 WA,占据原 38 的位置。由于 61>38,所以 61 可以排在第 1 个有序小文件中,故设置其段号为 1。以此类推,直到内存 WA 中所有的关键字都小于外存文件 FO 的最后一个记录的关键字(即段号都为 2),第 1 个初始归并段完成。开始建立第 2 个初始归并段,直到 FI 的所有记录都存到 FO 的初始归并段中。

algo10-4.cpp 是以教科书中图 11.5 的数据为例采用置换-选择排序的方法产生初始归并段文件的例子。按照一般的方法,如果内存最多只能存放 6 个记录,则 24 个记录要产生 4 个初始归并段文件。而采用置换-选择排序的方法只产生 3 个初始归并段文件。

无序的大数据文件 f10-2.txt 的内容如下：

```
51 1
49 2
39 3
46 4
38 5
29 6
14 7
61 8
15 9
30 10
1 11
48 12
52 13
3 14
63 15
27 16
4 17
13 18
89 19
24 20
46 21
58 22
33 23
76 24
```

```
// algo10-4.cpp 通过置换-选择排序产生不等长的初始归并段文件
#include"c1.h"
typedef int KeyType; // 定义关键字的类型为整型
typedef int InfoType; // 定义其他数据项的类型为整型
#include"c9-1.h" // 记录的数据类型
#define MAX_KEY INT_MAX // 最大关键字
#define w 6 // 设内存工作区可容纳的记录个数
#define M 10 // 设输出 M 个数据换行
typedef int LoserTree[w];
// [1..w-1]是败者树的非叶子结点,[0]中是胜者,存相应叶子的序号
typedef struct
{ RedType rec; // 记录(见图 10-1)
  int rnum; // 所属归并段的段号
}WorkArea[w]; // 内存工作区,容量为 w(见图 10-4)
void InputFromFile(FILE* f,RedType &c)
{ // 从文件输入记录的函数
  fscanf(f,"%d%d",&c.key,&c.otherinfo);
}
```

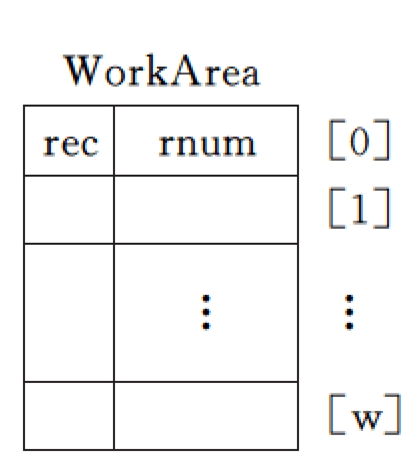


图 10-4　内存工作区类型

```
void OutputToFile(FILE * f,RedType &c)
{ // 向文件输出记录的函数
  fprintf(f,"%d %d\n",c.key,c.otherinfo);
}
void Select_MiniMax(LoserTree ls,WorkArea wa,int q) // 算法 11.6
{ // 从 wa[q]起到败者树的根比较选择当前段的最小记录给 ls[0]
  int s,p,t = (w + q)/2; // t 是败者树上 wa[q]的双亲结点序号
  for(p = ls[t];t>0;t = t/2,p = ls[t]) // 由 wa[q]的双亲结点逐级向根结点比较
    if(wa[p].rnum<wa[q].rnum||wa[p].rnum == wa[q].rnum&&
    wa[p].rec.key<wa[q].rec.key) // wa[q]的双亲结点的段值小或段值同关键字小
    { s = q; // wa[q]作为新的败者
      q = ls[t]; // q 指示新的胜利者,继续向上比较
      ls[t] = s;
    }
  ls[0] = q; // 最终胜利者的序号赋给 ls[0]
}
void Construct_Loser(LoserTree ls,WorkArea wa,FILE * fi)
{ // 输入 w 个记录到内存工作区 wa,建得败者树 ls,选出关键字最小的记录并由 s[0]指示
  // 其在 wa 中的位置。修改算法 11.7
  int i;
  for(i = 0;i<w;++i)
    wa[i].rnum = ls[i] = 0; // 初始化,工作区的段值为 0(最小),败者树指示最后调整的 wa[0]
  for(i = w - 1;i>= 0;-- i) // 从后到前
  { InputFromFile(fi,wa[i].rec); // 由文件输入 1 个记录
    wa[i].rnum = 1; // 其段号为"1"
    Select_MiniMax(ls,wa,i); // 根据新输入的记录调整败者树
  }
}
void get_run(LoserTree ls,WorkArea wa,int rc,int &rmax,FILE * fi,FILE * fo)
{ // 求得一个初始归并段,fi 为输入文件指针,fo 为输出文件指针,rc 为当前段。修改算法 11.5
  int q;
  KeyType minimax;
  while(wa[ls[0]].rnum == rc) // 选得的记录属当前段时
  { q = ls[0]; // q 指示选得的记录在 wa[]中的位置
    minimax = wa[q].rec.key; // minimax 指示选得的记录的关键字
    OutputToFile(fo,wa[q].rec); // 将选得的记录写入输出文件
    printf("(%d,%d)",wa[q].rec.key,wa[q].rec.otherinfo); // 输出
    InputFromFile(fi,wa[q].rec); // 从输入文件读入下一记录,填补输出的空位
    if(feof(fi)) // 输入文件结束
      wa[q].rnum = rmax + 1; // 虚设记录为下一段(属"rmax + 1"段)
    else // 输入文件非空时
    { if(wa[q].rec.key<minimax)
      { // 新读入记录的关键字小于刚输出到文件的记录的关键字
```

```
        rmax = rc + 1; // 设置下一段
        wa[q].rnum = rmax; // 新读入的记录属下一段
      }
      else // 新读入的记录属当前段
        wa[q].rnum = rc;
    }
    Select_MiniMax(ls,wa,q); // 从 wa[q]起选择新的当段最小关键字的记录
  }
}
void Replace_Selection(LoserTree ls,WorkArea wa,FILE * fi)
{ // 在败者树 ls 和内存工作区 wa 上用置换-选择排序求初始归并段,
  // fi 为已打开的输入文件(只读文件)指针。修改算法 11.4
  int rc,rmax;
  FILE * fo; // 输出文件指针
  char filename[3]; // 有序小文件名
  Construct_Loser(ls,wa,fi); // 初建败者树
  rc = rmax = 1; // rc 指示当前生成的初始归并段的段号,初值为 1
  do // rmax 指示 wa 中关键字所属初始归并段的最大段号,初值为 1
  { itoa(rc - 1,filename,10); // rc - 1 作为临时的有序小文件的文件名
    fo = fopen(filename,"w"); // 以写的方式打开输出文件 0,1,…
    printf("%s 的内容为:",filename);
    get_run(ls,wa,rc,rmax,fi,fo); // 求得段号为 rc 的初始归并段文件
    printf("\n");
    fclose(fo); // 关闭输出文件
    rc = wa[ls[0]].rnum; // 设置下一段的段号
  }while(rc<= rmax); // "rc = rmax + 1"标志输入文件的置换-选择排序已完成
  printf("共产生 %d 个初始归并段文件\n",rc - 1);
}
void main()
{
  FILE * fi;
  LoserTree ls;
  WorkArea wa;
  fi = fopen("f10-2.txt","r"); // 以读的方式打开无序大文件 f10-2.txt
  Replace_Selection(ls,wa,fi); // 用置换-选择排序求初始归并段(有序小文件)
  fclose(fi); // 关闭无序大文件 f10-2.txt
}
```

**程序运行结果**(以教科书中图 11.5 的数据为例,见图 10-5 和图 10-6):

```
0 的内容为:(29,6)(38,5)(39,3)(46,4)(49,2)(51,1)(61,8)
1 的内容为:(1,11)(3,14)(14,7)(15,9)(27,16)(30,10)(48,12)(52,13)(63,15)(89,19)
2 的内容为:(4,17)(13,18)(24,20)(33,23)(46,21)(58,22)(76,24)
共产生 3 个初始归并段文件
```

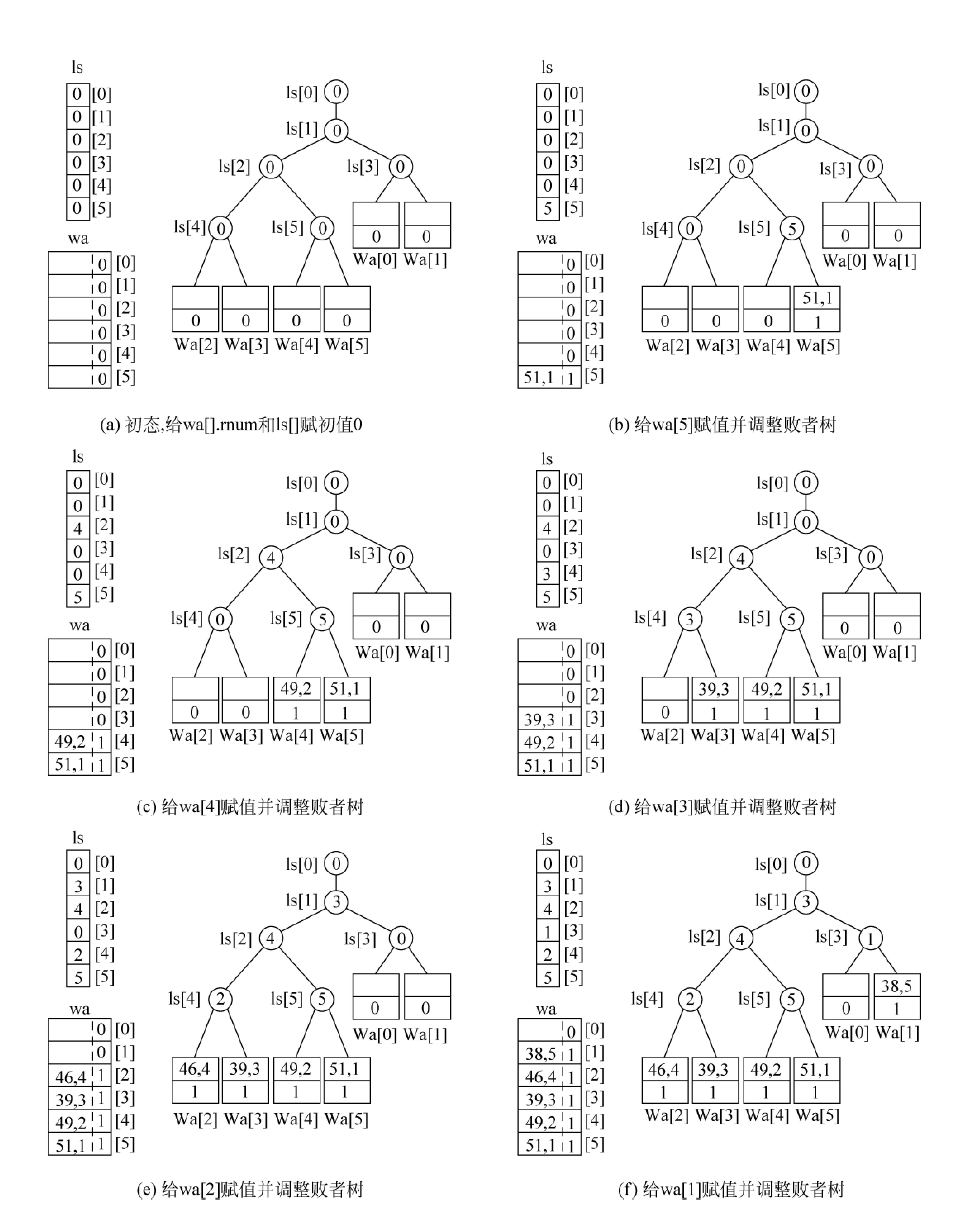
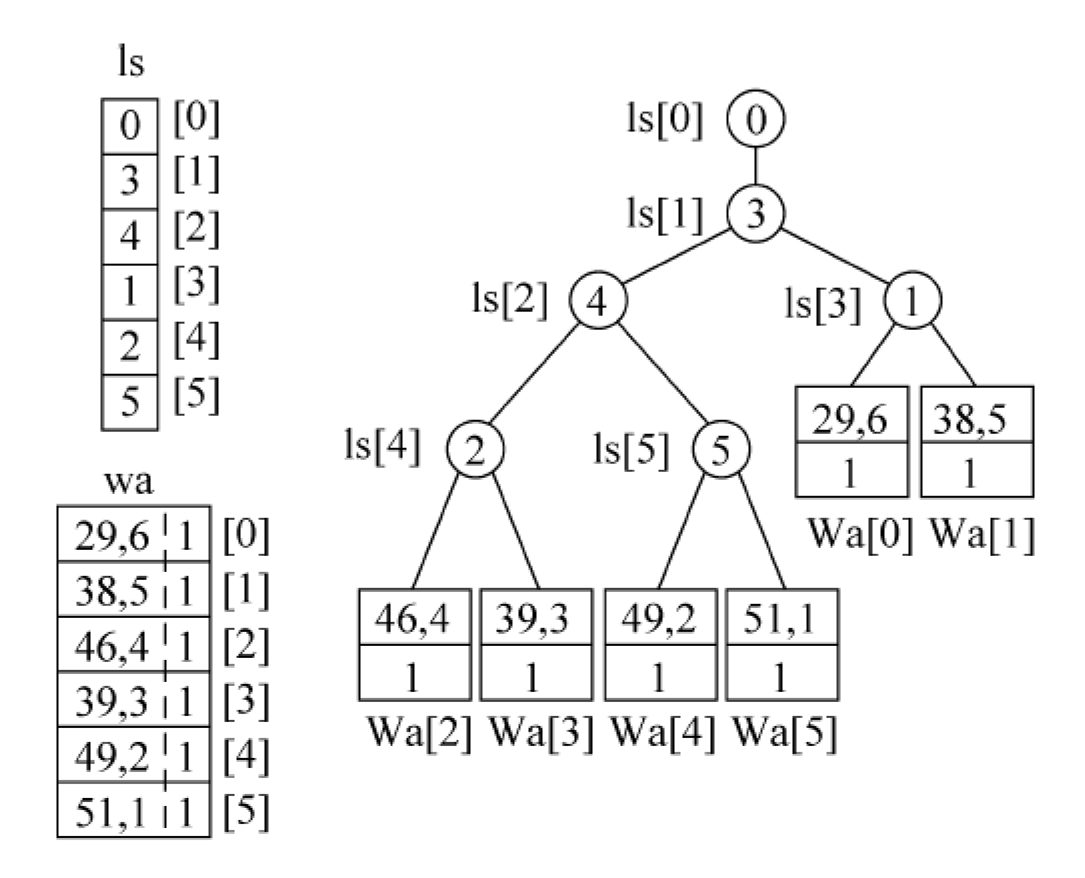



图 10-5　调用 Construct_Loser ()初建败者树示例




(g) 给wa[0]赋值并调整败者树，胜出者的序号在ls[0]

图 10-5 （续）

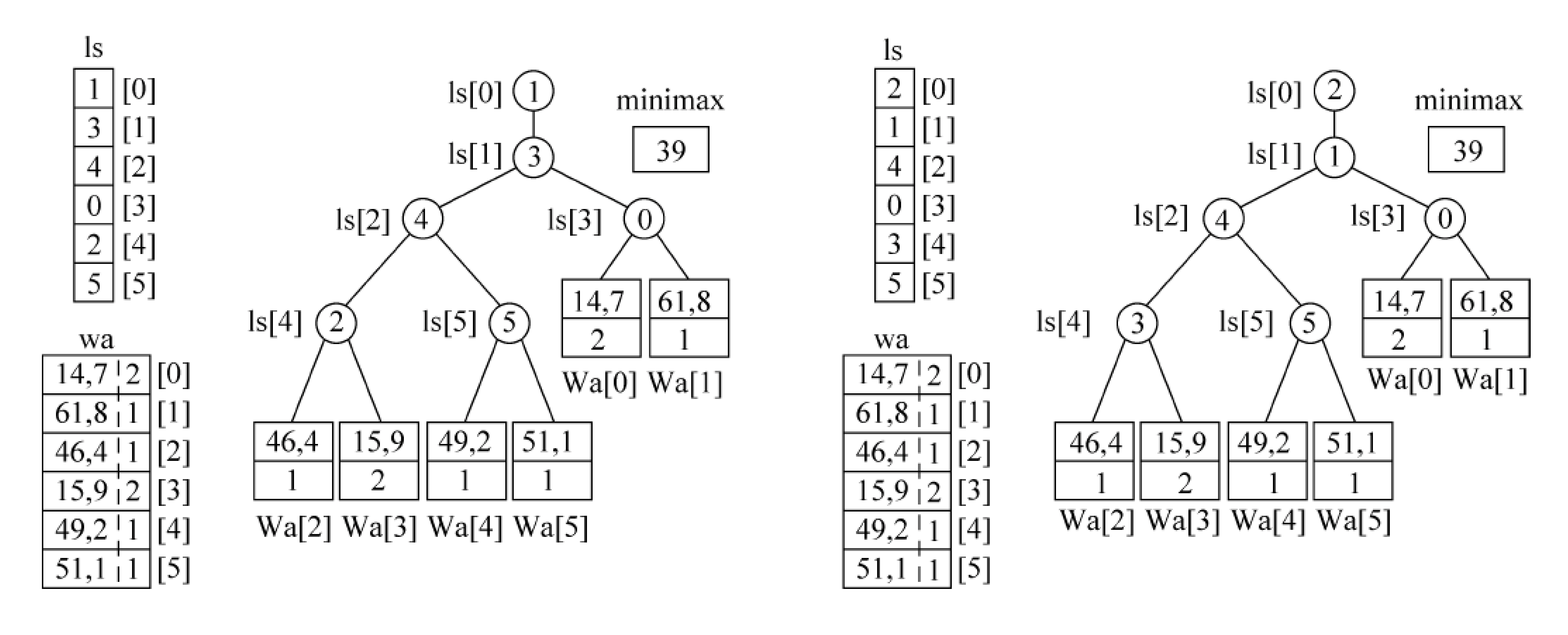
(a) 输出wa[0],再赋新值给wa[0]

(b) 从wa[0]起调整败者树

(c) 输出wa[1],再赋新值给wa[1]

(d) 从wa[1]起调整败者树

图 10-6　调用 get_run()示例



(e) 输出wa[3],再赋新值给wa[3]　　(f) 从wa[3]起调整败者树

图 10-6 （续）

运行 algo10-4. cpp,生成 3 个文件名分别为 0、1、2 的有序子文件,它们的内容如图 10-7 所示。将程序 algo10-3. cpp 的第 6 行作为注释,启用第 7 行,再运行 algo10-3. cpp,结果如下。

| 0 | 1 | 2 |
|---|---|---|
| 29 6 | 1 11 | 4 17 |
| 38 5 | 3 14 | 13 18 |
| 39 3 | 14 7 | 24 20 |
| 46 4 | 15 9 | 33 23 |
| 49 2 | 27 16 | 46 21 |
| 51 1 | 30 10 | 58 22 |
| 61 8 | 48 12 | 76 24 |
| | 52 13 | |
| | 63 15 | |
| | 89 19 | |

图 10-7　3 个有序子文件

**程序运行结果：**

```
请输入排序后的大文件名：f10-4.txt↙
有序大文件 f10-4.txt 的记录为：
(1,11)(3,14)(4,17)(13,18)(14,7)(15,9)(24,20)(27,16)(29,6)(30,10)
(33,23)(38,5)(39,3)(46,4)(46,21)(48,12)(49,2)(51,1)(52,13)(58,22)
(61,8)(63,15)(76,24)(89,19)
```

运行 algo10-3. cpp 产生的有序的大文件 f10-4. txt 的内容如下：

```
1 11
3 14
4 17
13 18
```



```
14 7
15 9
24 20
27 16
29 6
30 10
33 23
38 5
39 3
46 4
46 21
48 12
49 2
51 1
52 13
58 22
61 8
63 15
76 24
89 19
```

需要指出的是，程序 algo10-3. cpp 和 algo10-4. cpp 只是实现多路平衡归并和置换-选择排序的示例。为了提高效率、减少 I/O 操作，由文件向内存读数据和把内存数据写到文件还应以“数据块”为单位进行。以上两程序为了突出算法，由文件向内存读数据和把内存数据写到文件都是以“一个数据”为单位进行的。

# 附录A

## 关于标准C程序

本书中的程序一般需要经过修改才能在标准 C 的环境下运行，主要有以下 5 个原因。

(1) 教科书中的算法采用了 C++ 语言的引用调用的参数传递方式。在形参表中，以 & 开头的参数即为引用参数。

标准 C 不支持引用参数，对此需进行转换。下面以 bo1-1. cpp 中 DestroyTriplet()函数为例来说明这种转换。bo1-1. cpp 中含有引用参数的函数如下：

```
Status DestroyTriplet(Triplet &T)
{  free(T);
   T = NULL;
   return OK;
}
```

转换成能在标准 C 的环境下运行的函数如下：

```
Status DestroyTriplet(Triplet  * T) / * 将形参 &T 改为 * T * /
{ free( * T); / *  将函数体中的 T 改为 * T  * /
   * T = NULL; / *  将函数体中的 T 改为 * T  * /
   return OK;
}
```

对照以上 2 个函数可见：将 C++ 函数形参表中以 & 开头的参数改成以 * 开头的参数，再在函数体中该参数前加 * 即可。要注意的是，在这两个函数中，形参的类型是不同的。在 bo1-1. cpp 中，T 的类型是 Triplet；在转换后的函数中，T 的类型是 Triplet 的指针。另外，在标准 C 程序中调用该函数，实参前应加 &。如 main1-1. cpp 中调用 DestroyTriplet()的语句为

```
DestroyTriplet(T);
```

相应地，在标准 C 程序中调用 DestroyTriplet()的语句为

```
DestroyTriplet(&T); / *  实参前加 &  * /
```

其中，在调用 DestroyTriplet()的两程序中，两实参 T 的类型是相同的。另外，在转换过程中，遇到 & * 或 * & 可“抵消”，即将 * & T 转换为 T。

(2) 标准 C 在指明所定义的结构体或枚举类型时，在类型前要加 struct 或 enum，而

C++则不必加。例如，在 c2-1.h 中定义结构体 SqList 如下：

```
struct SqList
{ ElemType * elem;
  int length;
  int listsize;
};
```

C++在指明变量或形参的类型时，只须用 SqList 即可。例如，bo2-1.cpp 中的一个函数如下：

```
int ListLength(SqList L)
{ return L.length;
}
```

如果在标准 C 程序中像 c2-1.h 那样定义变量 L 的类型，则应修改以上函数如下：

```
int ListLength(struct SqList L) /* SqList 前加 struct */
{ return L.length;
}
```

说明形参 L 时要用 struct SqList。

当用到某个结构体就要在其类型前加 struct，用到某个枚举类型就要在其类型前加 enum 是很麻烦的。为了方便起见，可用 typedef 定义类型。在标准 C 程序中定义 SqList 如下：

```
typedef struct /* struct 前加 typedef */
{ ElemType * elem;
  int length;
  int listsize;
}SqList; /* 结构体名在此处 */
```

这样，bo2-1.cpp 不必做任何修改就可以用在标准 C 程序中了。

可见，只要在定义结构体时使用 typedef，则在指明结构体的类型时就不必加 struct。定义枚举类型时使用 typedef 的方法与此类似。

(3) 标准 C 的共用体必须有变量名，而 C++可以省略。例如，在 c5-5.h 中定义 GLNode1 如下：

```
typedef struct GLNode1
{ ElemTag tag;
  union
  { AtomType atom;
    GLNode1 * hp;
  }; // 共用体可以没有变量名
  GLNode1 * tp;
} * GList1,GLNode1;
```

**注意**：其中的共用体 union 没有变量名。

bo5-6.cpp 中的函数 GListEmpty()如下：

```
Status GListEmpty(GList1 L)
{ if(!L->hp)
    return OK;
  else
    return ERROR;
}
```

而在标准 C 下，对 c5-5.h 要作如下修改：

```
typedef struct GLNode1
{ ElemTag tag;
  union
  { AtomType atom;
    struct GLNode1 *hp;
  }a; /* 给共用体加变量名 */
  struct GLNode1 *tp;
} *GList1,GLNode1;
```

GLNode1 内部的共用体必须有变量名。bo5-6.cpp 中的函数 GListEmpty()也要作相应修改：

```
Status GListEmpty(GList1 L)
{ if(!L->a.hp) /* hp 前加 a. */
    return OK;
  else
    return ERROR;
}
```

(4) C++可重载。在一个程序中，可有几个同名的函数同时存在，只要它们的形参个数或类型有所不同即可。标准 C 不可重载。在 C++转换成标准 C 时，必须将同名函数改为不同名。如在 bo8-2.cpp 中有 1 个 SearchBST()函数(算法 9.5(b))，在 func8-4.cpp 中也有 1 个 SearchBST()函数(算法 9.5(a))，它们都被 algo8-4.cpp 调用。但这两个同名函数的形参个数不同。C++根据函数的形参个数可分辨所调用的是哪个函数，而标准 C 没有这个能力，所以在标准 C 中要将其中的 1 个 SearchBST()函数改名为 SearchBST1()函数。

(5) 在 C++中“ // ”后到本行末的内容为注释，而标准 C 的注释必须放在“/ * ”、“ * /”之间。如上许多例子所示。

# 参 考 文 献

[1] 严蔚敏，吴伟民. 数据结构(C 语言版). 北京：清华大学出版社，2002

[2] (美)布莱斯. 数据结构与算法——面向对象的 C++ 设计模式. 胡广斌等译. 北京：电子工业出版社，2003

[3] 高一凡. 数据结构算法实现及解析——配合严蔚敏，吴伟民编著的《数据结构》(C 语言版). 第 2 版. 西安：西安电子科技大学出版社，2004

# 读者意见反馈

亲爱的读者：

感谢您一直以来对清华版计算机教材的支持和爱护。为了今后为您提供更优秀的教材，请您抽出宝贵的时间来填写下面的意见反馈表，以便我们更好地对本教材做进一步改进。同时如果您在使用本教材的过程中遇到了什么问题，或者有什么好的建议，也请您来信告诉我们。

地址：北京市海淀区双清路学研大厦 A 座 602　计算机与信息分社营销室　收

邮编：100084　　电子邮箱：jsjjc@tup.tsinghua.edu.cn

电话：010-62770175-4608/4409　　邮购电话：010-62786544

教材名称：数据结构算法解析

ISBN：978-7-302-15879-0

**个人资料**

姓名：__________　年龄：________所在院校/专业：________________

文化程度：________　通信地址：________________________

联系电话：________　电子信箱：________________________

**您使用本书是作为：**□指定教材 □选用教材 □辅导教材 □自学教材

**您对本书封面设计的满意度：**

□很满意 □满意 □一般 □不满意　改进建议____________________

**您对本书印刷质量的满意度：**

□很满意 □满意 □一般 □不满意　改进建议____________________

**您对本书的总体满意度：**

从语言质量角度看　□很满意 □满意 □一般 □不满意

从科技含量角度看　□很满意 □满意 □一般 □不满意

**本书最令您满意的是：**

□指导明确 □内容充实 □讲解详尽 □实例丰富

**您认为本书在哪些地方应进行修改？（可附页）**

________________________________________

________________________________________

**您希望本书在哪些方面进行改进？（可附页）**

________________________________________

________________________________________

# 高等学校教材·计算机科学与技术
## 系列书目

| 书　号 | 书　名 | 作　者 |
|---|---|---|
| 9787302103400 | C++程序设计与应用开发 | 朱振元等 |
| 9787302135074 | C++语言程序设计教程 | 杨进才等 |
| 9787302140962 | C++语言程序设计教程习题解答与实验指导 | 杨进才等 |
| 9787302124412 | C语言程序设计教程习题解答与实验指导 | 王敬华等 |
| 9787302091301 | Java面向对象程序设计教程 | 李发致 |
| 9787302133957 | Visual C#.NET程序设计教程 | 邱锦伦等 |
| 9787302118565 | Visual C++面向对象程序设计教程与实验 | 温秀梅等 |
| 9787302112952 | Windows系统安全原理与技术 | 薛质 |
| 9787302133940 | 奔腾计算机体系结构 | 杨厚俊等 |
| 9787302098409 | 操作系统实验指导——基于Linux内核 | 徐虹等 |
| 9787302097648 | 程序设计方法解析——Java描述 | 沈军等 |
| 9787302086451 | 汇编语言程序设计教程 | 卜艳萍等 |
| 9787302092568 | 计算机导论 | 袁方等 |
| 9787302137801 | 计算机控制——基于MATLAB实现 | 肖诗松等 |
| 9787302116134 | 计算机图形学原理及算法教程（Visual C++版） | 和青芳 |
| 9787302137108 | 计算机网络——原理、应用和实现 | 王卫亚等 |
| 9787302126539 | 计算机网络安全 | 刘远生等 |
| 9787302118664 | 计算机网络基础教程 | 康辉 |
| 9787302139201 | 计算机系统结构 | 周立等 |
| 9787302134398 | 计算机原理简明教程 | 王铁峰等 |
| 9787302111467 | 计算机组成原理教程 | 张代远 |
| 9787302130666 | 离散数学 | 李俊锋等 |
| 9787302104292 | 人工智能（AI）程序设计（面向对象语言） | 雷英杰等 |
| 9787302141006 | 人工智能教程 | 金聪等 |
| 9787302136064 | 人工智能与专家系统导论 | 马鸣远 |
| 9787302093442 | 人机交互技术——原理与应用 | 孟祥旭等 |
| 9787302129066 | 软件工程 | 叶俊民 |
| 9787302117186 | 数据结构——Java语言描述 | 朱战立 |
| 9787302093589 | 数据结构（C语言描述） | 徐孝凯等 |
| 9787302093596 | 数据结构（C语言描述）学习指导与习题解答 | 徐孝凯等 |
| 9787302079606 | 数据结构（面向对象语言描述） | 朱振元等 |
| 9787302099840 | 数据结构教程 | 李春葆 |
| 9787302108269 | 数据结构教程上机实验指导 | 李春葆 |

| 书　　号 | 书　　名 | 作　　者 |
| --- | --- | --- |
| 9787302108634 | 数据结构教程学习指导 | 李春葆 |
| 9787302112518 | 数据库系统与应用（SQL Server） | 赵致格 |
| 9787302106319 | 数据挖掘原理与算法 | 毛国君 |
| 9787302126492 | 数字图像处理与分析 | 龚声蓉 |
| 9787302124375 | 算法设计与分析 | 吕国英 |
| 9787302103653 | 算法与数据结构 | 陈媛 |
| 9787302136767 | 网络编程技术及应用 | 谭献海 |
| 9787302071310 | 微处理器（CPU）的结构与性能 | 易建勋 |
| 9787302109013 | 微机原理、汇编与接口技术 | 朱定华 |
| 9787302140689 | 微机原理、汇编与接口技术学习指导 | 朱定华 |
| 9787302128250 | 微机原理与接口技术 | 郭兰英 |
| 9787302084471 | 信息安全数学基础 | 陈恭亮 |
| 9787302128793 | 信息对抗与网络安全 | 贺雪晨 |
| 9787302112358 | 组合理论及其应用 | 李凡长 |